# Understanding & Analyzing the Garbage-First Family of Garbage Collectors

**Wenyu Zhao**

A thesis submitted for the degree of
Bachelor of Compiter Science with Honours
The Australian National University

September 2018

Except where otherwise indicated, this thesis is my own original work.

Wenyu Zhao
29 September 2018

to my xxx, yyy (yyy is the people you want to dedicated this thesis to.)

# Acknowledgments

Who do you want to thank?

# Abstract

The Garbage-First family of garbage collectors, including Garbage-first GC, Shenandoah GC and ZGC, are region-based concurrent collectors designed for large heaps to archive lower latency and mutator overhead. However, analysis, measurements and comparisons between their OpenJDK implementations can be hard since the implementations do not share too much code and have different level of optimizations. This article attempts to perform a detailed and careful analysis of these collectors at an algorithmic level by implementing them upon the same hierarchy on JikesRVM. We evaluate the footprint overheads, mutator latency and barrier overheads for these collectors. Our results show that by using remembered-sets, G1 has ???% average footprint overhead and G1, Shenandoah has barrier overheads of ???% and ???% respectively on the DaCapo benchmark suite. We also find that by using Brooks style indirection pointers, Shenandoah and ZGC has average barrier overheads of ???% and ???%. The costs and overheads revealed in these garbage collectors can help garbage collection algorithm designers to identify the main advantages and drawbacks of region-based concurrent collectors and hence have the ability to make further refinements and optimizations to them. In additon, the relationships this paper discussed in the

# Contents

# List of Figures

# List of Tables

Chapter 1

# Introduction

## 1.1 Motivation

As the memory size of modern server machines becomes larger, latency time of garbage collectors for managed programming languages generally becomes longer. In this way, the designing of low-latency garbage collectors has become a hot topic today. This involves reducing the latency of collectors and performing heap compaction or evacuation to avoid heap fragmentation.

As the three major concurrent region-based garbage collectors in OpenJDK, Garbage-First GC and Shenandoah GC share most of their collection policies and techniques. Both of these two GCs were designed to deduce latency of mutators on large heaps. They share the same heap structure which divides the heap up into regions. During GC cycle, both of these two GCs perform similar marking and heap compaction processes. Although the specific algorithms and implementation details are very, these two GCs reveal strong relationships to each other and share plenty of common parts and general ideas.

These Garbage-First family of garbage collectors were designed to reduce GC latencies, their performances sometimes can be unsatisfactory. Sometimes GC time of region-based collectors can take up to 50% of the total execution time of Big Data systems (BrianDemsky [2015]). In this way it is necessary to perform careful analysis of the pros and cons for the Garbage-First family of garbage collectors, especially on an algorithmic level.

However, analyzing the original implementations of these three garbage collectors on OpenJDK is difficult. These three collectors are implemented by different teams and do not share too much code among the implementations. Also, since Garbage-First is production ready while Shenandoah GC is still provided as an experimental GC option in OpenJDK, optimizations among their implementations varies a lot. Thus analyzing the original OpenJDK implementations becomes impossible.

This thesis aims to perform an more careful analysis and comparison among all the Garbage-First family of garbage collectors by implementing these collectors on JikesRVM and sharing as much implementation as possible.

## 1.2 Approach

In order to make detailed and careful analysis on these collectors, we started from implementing a simple stop-the-world and region-based GC, and made progressive refinements on it to implement a linear-scan evacuation GC, a concurrent-marking GC, the Garbage-First GC and Shenandoah GC. All of these garbage collectors share the same implementation of the region-based heap structure, allocators and concurrent marking phase (excepted for the stop-the-world region-based GC). In order to compare these collectors at an algorithmic level, the implementations are as close as possible to the design of their original papers.

We evaluated the footprint overhead for the remembered-sets of the Garbage-First GC. We also evaluated the Brooks barriers overhead for Shenandoah GC, the latency time and the snapshot-at-the-beginning (SATB) barrier time for all collectors on the DaCapo benchmark suite.

All work were done in JikRVM, a research purpose Java virtual machine and MMTk, a memory management took written in java (Blackburn et al. [2004]).

## 1.3 Contribution

Five different Garbage-First family of garbage collectors were implemented to reveal the relationships among them and demonstrate different kinds of garbage collection algorithms, such as concurrent marking, remembered-sets and Brooks barriers.

Detailed and careful analysis on barrier latency and GC pause time was made to assist with comparisons among these collectors. We found that concurrent marking reduces the average GC pause time by ???% where linear scanning evacuation can increase the average GC pause time by ???%. However, the use of card scanning can reduces the GC pause time up to ???%. As the result of barrier latency analysis, we found that SATB barriers can increase the mutator overhead by ???% where Brooks barriers increase the mutator overhead of Shenandoah by ???%.

## 1.4 Thesis Outline

Chapter 2 provides an general background and an overview of the Garbage-First family of garbage collectors. Similarities and differences among current collectors in OpenJDK are discussed, especially those region-based collectors.

Chapter **??** roughly describes the platform details where the all Garbage-First family of garbage collectors were implemented and evaluated.

Chapter 4.2 provides the detailed steps and algorithms of the implementation of all the Garbage-First family of garbage collectors.

Chapter **??** 5 describes the methodology of testing and evaluating the implemented

collectors. Including the benchmark used and the detailed steps of the evaluation.

Chapter 6 presents the results of the evaluation and benchmarking on the implemented collectors, as well as results from some other collectors in MMTk as a comparison.

Chapter 7 performs detailed and critical discussion on the evaluation results of the implemented collectors. Some recommendations for algorithm improvements are also provided.

Chapter 2

# Background and Related Work

This chapter describes the background and basic ideas of several garbage collectors, particularly those targeting the java virtual machine and are implemented in OpenJDK or JikesRVM, as well as the differences among several garbage collection techniques.

In addition, this chapter also performed a general discussion of the related work on analyzing region-based garbage collectors.

Section 2.2 roughly discusses and compares different class of GC algorithms.

Section 2.3 describes the general design of the Garbage-first family of garbage collectors.

Section 2.4 describes the related work on analyzing region-based garbage collectors.

## 2.1 JikesRVM and MMTk

The whole project discussed in this thesis are based on JikesRVM and all the garbage collectors we implemented and evaluated in this project are all implemented by using MMTk. In this section we will briefly discuss the design of JikesRVM and MMTk, as well as some description on the general structure of them.

### 2.1.1 JikesRVM

JikesRVM is a research purpose Java virtual machine. It is a meta-circular JVM which is implemented in the Java programming language and is self hosted. JikesRVM was designed to provide a flexible open sourced test-bed to experiment with virtual machine related algorithms and technologies.

In stead of executing Java programs by directly interpreting the Java byte code, JikesRVM compiles them into machine code for execution. JikesRVM implemented two tiers of compilers, the baseline compiler and the optimizing compiler. The

baseline compiler simply translates the Java byte code into machine code and do no optimizations while the optimizing compiler performs several optimizations during the code generation phase. We performed all benchmarking and analysis works on the optimized build of the JikesRVM.

### 2.1.2 MMTk

MMTk is a memory management toolkit. MMTk is used as a memory management module for JikesRVM and is responsible for memory allocation and garbage collection.

For memory allocation, MMTk defines several address spaces to allocate different type of objects. e.g. `NonMovingSpace` for non-copyable objects and `SmallCodeSpace` for storing java code. After receving a allocation request, MMTk will decide which space the object belongs to and allocate memory from that space.

All of the G1 family of garbage collectors involved in this thesis involves two major phases, the marking phase and the evecuation phase. Instead of using the MMTk's pre-defined `PREPARE -> SCAN ROOTS -> CLOSURE -> RELEASE` collection phase which only performs a single tracing on the heap, we extended this to perform a seperate full or partial heap tracing or linear scanning phase for evacuation and reference updating.

MMTk will check for stop-the-world or concurrent garbage collection after each requested space allocation. This involves the invocation of methods `collectionRequired(...)` and `concurrentCollectionRequired(...)`. We made full use of these two methods, not only for checking whether a collection is required, but also performs switching between different schedule of collection phases for either nursery or mature collection for the G1 collector.

## 2.2 Categories of GC Algorithms

This section discusses the main classes of garbage collection algorithms, as well as their pros and cons.

### 2.2.1 Reference counting

Reference counting is a widely used garbage collection technique which tracks the count of references for each heap allocated object (Detlefs et al. [2002]). The reference count for a object is increased when a new variable references to the object and decreased when a reference to the object is deleted or goes out of its declaring scope. The reference count for a object is initialized to one when the object is created, which means there is only one owner for the object (its creater). When the reference count of the object goes to zero, it is certain that the object has no owner that references to it. Then the object becomes floating garbage and its occupying memory is released.

In order to track the reference count for each object, a write barrier is involved for reference counting collectors. For each object reference modification `obj.x = y`, the

reference count of the old object reference is decreased and the reference count of the new object reference is increased by 1. If the old object reference has no owner, then its memory cell gets sweeped immediately.

This collection algorithm is highly efficient and the workload for sweeping objects are almost evenly distributed which makes the mutator pause time for reference counting to be extremely short.

However, one major disadvantage of the reference counting GC is that it can hardly handle cyclic references (Lins [1992]) where some object `A` references the other object `B` and `B` also references `A`. In such case the reference count is at least 1 for both `A` and `B`, even if there is no objects referencing to them.

### 2.2.2 Mark & Sweep GC

Mark and Sweep GC is a type of tracing GC which uses the object graph to assist with garbage collection. The algorithm considers all the objects that is the unreachable in the program as garbage (Endo et al. [1997]). In this way Mark and Sweep GC has the ability to collect cyclic referenced garbages, as long as they are unreachable from other live objects.

When allocating objects or requesting memory pages from some memory resource, the mutators dose no extra work but only checks whether the memory is full. If there is no free space for allocation, the execution of all the mutators will be paused and the Mark and Sweep GC is triggered.

The Mark and Sweep GC requires an extra metadata byte in the object header for marking. During each GC cycle, the Mark and Sweep GC firstly scans and marks all the static object, global objects and stack variables as a set of "root objects". Then the collector starting from the root objects and recursively walks over the object graph to mark all the remaining objects. At the end of the marking phase, all the marked objects in the heap are reachable from the "root set" and all other objects become floating garbage and are sweeped.

After a GC cycle, all the paused mutators are resumed to continue execution and allocation.

The Mark and Sweep GC has the ability to collect cyclic referenced garbages by tracing the reachability of heap objects. However, as the use of large-scaled servers and other programs for business becomes more and more porpular, the Mark and Sweep GC reveals its drawback that it can cause significant memory fragmentation after a sufficiently long running time. Because when the collector keeps allocate and free small memory chunks, the size of contiguous free chunks becomes smaller, which may leads to allocation failure for large objects, even if the total free memory size is larger than the requested chunk size.

### 2.2.3 Copying GC

Copying GC is a class of garbage collectors that aims to reduce heap fragmentation by performing heap compation, which moves objects in the heap together.

As one of the most simple copying GC, the SemiSpace GC divides the whole heap into two spaces, the from-space and the to-space. All objects allocation are done within the from-space. When the from-space is exhausted, a GC cycle is triggered. The collector will starting from all root objects, and recursively walks over the object graph to copy all reachable objects to the to-space. Then the to-space becomes the new from-space for further allocation. In this way the SemiSpace GC ensures all live objects are copied to the to-space and all non-reachable objects (i.e. dead objects) are not forwarded and are sweeped at the end of the GC cycle.

Copying GCs has the ability to reduce heap fragmentation but can cause longer pause time for each GC cycle. Especially for some GCs which has additional evacuation phase at the end of a marking phase, e.g. the MarkCompact GC.

## 2.3 Garbage-First Family of Garbage Collectors

This section will give a brief introduction the three most popular Garbage-first family of garbage collectors. For each collector, the basic algorithm and the past and current status will be discussed.

### 2.3.1 Garbage-First GC

Garbage-First GC is a copying collector which was initially released in Oracle JDK 6 and was fully supported in Oracle JDK 7. G1 was designed as server-style collector which targeting machines with multi-processors and large memories.

G1 GC divides the whole heap up into some fiexed sized regions. As a copying collector, G1 GC tries to reduce the pause time for evacuating objects by performing evacuation on a subset of regions (called the collection set) instead of all allocated regions.

The collection cycle of the Garbage-First GC starts with a concurrent marking phase which the collector threads marks all the objects in the heap, just like the Mark & Sweep GC, but without pausing the mutators. After the marking phase, the G1 collector selects the collection set which contains the regions with smallest ratio of live objects. Then the collector evacuates live objects in the collection set.

In order to perform evacuation on a subset of regions, the collector uses a data structure called "remembered set" to remember all uses of the objects in the collection set. After these live objects are evacuated, the collector scans the remembered set to update the pointers in other regions that references these live objects.

By performing partial heap evacuation, the G1 GC generally has lower pause time than other copying GCs, especially on machines with large heaps (Detlefs et al. [2004]). By adjusting the size of the collection set before evacuation, G1 has the ability to control the pause time to meet some user-defined soft pause time goal. However, one main drawback of G1 is that it is not suitable for small heaps, and the implementation of remembered sets can be inefficient.

### 2.3.2 Shenandoah GC

Shenandoah GC is concurrently an experimental collector for OpenJDK. Shenandoah GC also divides heap up into regions and performs concurrent marking, similar to the Garbage-first GC.

Shenandoah GC tries to further reduces the GC latency by performing concurrent compaction. The concurrent marking phase that Shenandoah GC has is similar to the G1's concurrent marking phase. However, Shenandoah GC does not have a generational mode and does not perform pertial evacuation to reduce pause times. Instead, the Shenandoah GC performs the evacuation phase concurrently to collect all posible regions, without pausing mutators.

By performing concurrent marking and compaction, Shenandoah GC does most of the heap scanning work in concurrent. In this way the pause time caused by garbage collection is extremely small and is not proportional to the heap size. However, Shenandoah GC has to insert some mutator barriers into the Java program, before every object reference read and write. So the mutator overhead caused by these barrier is much greater than other GCs.

### 2.3.3 ZGC

ZGC is very similar to the Shenandoah GC and is also currently under experimental status. ZGC performs a refinement on the Shenandoah GC to largely reduces the barrier overhead. ZGC assigns a color for each pointer and stores this metadata into the unused bits in the 64bit pointers. Instead of inserting barriers on every object reference read and write, ZGC only uses a "load barrier" which is only inserted before the mutator loads an object reference from the heap. The barrier is responsible for both object concurrent marking and evacuation, by checking the "color" metadata in the pointer. In this way ZGC remarkably reduces the throughput reduction caused by the mutator barriers.

## 2.4 Related Work

### 2.4.1 Implementations of the G1 family of collectors

All of the three major G1 family of collectors has been implemented in OpenJDK. Detlefs et al. [2004] demostrates the basic algorithm and the original design of the Garbage-First collector.

### 2.4.2 Evaluation of region-based collectors

You may reference other papers. For example: Generational garbage collection [Lieberman and Hewitt, 1983; Moon, 1984; Ungar, 1984] is perhaps the single most important advance in garbage collection since the first collectors were developed in the early 1960s. (doi: "doi" should just be the doi part, not the full URL, and it will be made

to link to dx.doi.org and resolve. shortname: gives an optional short name for a conference like PLDI '08.)

## 2.5 Summary

In this section we introduced the background and basic ideas of several garbage collectors and also performed a general discussion of the related work on analyzing region-based garbage collectors.

Next chapter will briefly describe the implementation environment of the collectors implemented in this thesis, including the high-level design of JikesRVM and MMTk.

Chapter 3

# Implementations and Refinements

The original Garbage-First GC has become the default garbage collectors since Java 9 while the Shenandoah GC and ZGC implementations are still not production ready. Due to such implementation status differences, the original implementations of these collectors are at a different optimization level and do not share too much code even they have similar marking barriers, heap structures and allocation policies.

In order to erase the implementation differences between these collectors and perform a careful and pure algorithmic level comparison and analysis among them, we implemented several Garbage-First family of garbage collectors on JikesRVM, by using the Memory Management Toolkit (MMTk).

In this chapter, we discuss the motivation of performing the implementations, the general design of the GC implementations, the implementation details for all the garbage collectors, each step of the progressive refinements as well as the relationships between these collectors.

Section 3.1 describes the general and high-level design of the implementation procedure.

Section 3.2 describes the implementation details of a simple region-based GC.

Section 3.3 describes the implementation of a refinement of the simple region-based GC that uses linear scanning during evacuation.

Section 3.4 describes the implementation of a refinement that performs object marking concurrently.

Section 3.5 describes the details of implementing the Garbage-First GC

Section 3.6 describes the details of implementing the Shenandoah GC

## 3.1 General Design

We started from implementing a simple region-based GC which divides the whole java heap up into fix sized regions and performs fully stop-the-world but parallel object marking and full-heap tracing based heap compaction/evacuation during each GC. We made progressive refinements based on this collector to further implement some G1 family of garbage collectors.

Since both Garbage-First and Shenandoah GC use heap linear scanning for evacuation, we implemented a linear-scan region-based GC by divided the heap evacuation phase of the simple region-based GC into two phases: The linear scan evacuation phase and the reference updating phase.

Then we changed the stop-the-world marking phase of the linear-scan region-based GC to the concurrent marking phase to implement a concurrent-marking region-based GC, by using the Snapshot at the Beginning algorithm (Yuasa [1990]).

By implementing the remembered-sets and remembered-set based evacuation based on the previous concurrent-marking region-based GC, we implemented the Garbage-First collector.

Also starting from the concurrent-marking region-based GC, by implementing the Brooks indirection pointers and the corresponding mutator barriors, we implemented the Shenandoah GC.

By performing such progressive refinements, we successfully implemented a series of G1 family of garbage collectors and share as much code and design as possible among the implementations of these collectors. Which enables the possibility for future algorithmic-level analysis on these garbage collectors.

## 3.2 Simple Region-based GC

This simple region-based GC is provided as a baseline for future refinements. It contains most basic structures of the Garbage-First family of garbage collectors such as region-based heap space and bump pointer allocation algorithm. By making progressive refinements on this region-based GC, we keep the implementation differences between the Garbage-First family of garbage collectors to a minimum.

### 3.2.1 Heap structure and allocation policy

This simple region-based GC and other GCs discussed later all uses the same implementation of the heap structure (which is implemented in `RegionSpace` and `Region` classes) and the same allocation policy (which is implemented in the `RegionAllocator` class). The `RegionSpace` divides the whole heap up into fixed 1 MB regions, which are 256 MMTk pages.

Figure 3.1 shows the code for allocation objects within the region space. To allocate objects, for each allocator, it firstly requests a region of memory (256 pages) from the page resource. Then it makes the allocation cursor points to the start of the region and bump increase this allocation cursor to allocate objects. Figure 3.1(a) shows the

```
@Inline
public final Address alloc(int bytes, int align, int offset) {
  /* establish how much we need */
  Address start = alignAllocationNoFill(cursor, align, offset);
  Address end = start.plus(bytes);
  /* check whether we've exceeded the limit */
  if (end.GT(limit)) {
    return allocSlowInline(bytes, align, offset);
  }
  /* sufficient memory is available, so we can finish performing the allocation
      */
  fillAlignmentGap(cursor, start);
  cursor = end;
  // Record the end cursor of this region
  Region.setCursor(currentRegion, cursor);
  return start;
}
```

(a) Region allocator - fast path

```
@Override
protected final Address allocSlowOnce(int bytes, int align, int offset) {
  // Acquire a new region
  Address ptr = space.getSpace(allocationKind);
  this.currentRegion = ptr;

  if (ptr.isZero()) {
    return ptr; // failed allocation --- we will need to GC
  }
  /* we have been given a clean block */
  cursor = ptr;
  limit = ptr.plus(Region.BYTES_IN_REGION);
  return alloc(bytes, align, offset);
}
```

(b) Region allocator - slow path

**Figure 3.1:** Region Allocator

code for such allocation fast path. Most of the object allocation processes will only go to the fast path. However after a region is filled, to allocate a new object, the mutator enters a slow path which is shown in figure 3.1(b) that moves the bump pointer to another newly acquired region for future allocations.

MMTk reserves some extra pages for each region to record metadata. After the allocator filled a region, it records the end address of the region for region linear scanning which is later used in some collectors. Also, an off-heap bitmap is maintained in the metadata pages of the region to record the liveness data of objects allocated in this region.

```
@Inline
public ObjectReference traceMarkObject(TransitiveClosure trace, ObjectReference
    object) {
  if (testAndMark(rtn)) {
    Address region = Region.of(object);
    // Atomically increase the live bytes of this reigon
    Region.updateRegionAliveSize(region, object);
    // Push into the object queue
    trace.processNode(object);
  }
  return object;
}
```

**Figure 3.2:** Code for mark each object

### 3.2.2 Stop-the-world marking

Based on the high-level design of MMTk, after each allocation MMTk checks if a stop-the-world or concurrent GC is required for for the current selected GC plan. For this simple region-based GC, only stop-the-world collections are triggered.

The region-based GC initiates a collection cycle when the free heap size is less than the pre-defined `reversedPercent` (default is 10%). During each GC cycle, the collector starts by performing a full heap tracing to recursively mark all live objects. The marking algorithm considers the heap as a graph of objects and follows the idea of Breadth-first graph search which firstly scan and mark all the stack and global root objects and pushes them into an object queue. Then the collectors threads keeps popping objects from the object queue, collects all its object reference fields (which are child nodes of the current object node in the object graph), scan these fields and if they are not marked previously, push them back into the object queue. The marking process is done when the global object queue is drained, which means all object that are reachable from the root objects are marked after the marking phase.

In stead of using the gc byte in the object header for marking which is widely used in other GCs in MMTk, we maintain a bitmap for each region to record the liveness data for each object within this region. At the start of the marking phase, bitmaps of all regions are initialized to zero. As shown in figure 3.2, during visiting each object, the collector attempts to set the mark bit of the current object in the bitmap, and push the object into the object queue only if the attempt is succeed.

Although using such extra "live table" is not necessary for this region-based GC, this is a common design for Garbage-First and Shenandoah GC. So we disable the gc byte in the object header at the very beginning to reduce the implementation difference among all collectors.

### 3.2.3 Collection set selection

As shown in figure 3.2, during the processing of each object in the heap, the collectors also atomically increase the live bytes for each region, starting from zero. After

the full heap marking phase, the collector starts a collection set selection phase to construct a collection set of regions.

The collection set selection phase firstly takes a list of all allocated regions, sort them in ascending order by the live bytes of the region. Then the collector selects the regions with lowest live bytes. Also the collector should make sure the total live bytes in the collection set is not greater than the free memory size of the heap.

At the end of the collection set selection phase, the collector marks all regions in the collection set as `"relocationRequired"` for the future evacuation phase.

### 3.2.4 Stop-the-world evacuation

The evacuation phase is a fundamental part of the copying collectors. It tries to avoid heap fragmentation by forwarding the live objects which are sparsely located in the heap and compacting them together.

For this simple region-based GC, the evacuation phase is performed after the end of the collection set selection phase. This evacuation phase is aiming to copy/evacuate all live objects in the collection set to other regions.

Figure 3.3 shows the process of object evacuation. To evacuate objects, the collector performs another full heap tracing to remark all the objects, just like the marking phase discussed before. But in addition to mark the objects, the collector also copy the objects and atomically set the forwarding status in the object header if the object is in the relocation set. The same full heap tracing ensures all objects marked in the marking phase are also being scanned in the evacuation phase. Which means all live objects in the collection set are processed and evacuated properly.

At the end of the evacuation phase, after all objects are evacuated, the whole stop-the-world GC cycle is finished. The collector frees all the regions in the collection set and resumes the execution of all the mutators.

### 3.2.5 Evacuation correctness verification

Since the region-based evacuation is a fundamental component of G1 family of garbage collectors and can have several variants, it is necessary to verify the correctness of the evacuation process. We used an additional full heap tracing for verification. The full heap tracing is similar to the marking trace and is fully stop the world to ensure the object graph is never changed during the verification process. When visiting each object node during verification, the collector checks all its object reference fields and ensure they are either null or are pointing to the valid java objects. The collector also checks no object node should be located in the from space (i.e. the collection set).

This full heap tracing verification process is optional and can be switched on and off for debugging purposes. At the end of the verification process, we can assert that the whole java heap is not broken and is in the correct status.

```
1  @Inline
2  public ObjectReference traceEvacuateObject(TraceLocal trace, ObjectReference
       object, int allocator) {
3    if (Region.relocationRequired(Region.of(object))) {
4      Word priorStatusWord = ForwardingWord.attemptToForward(object);
5      if (ForwardingWord.stateIsForwardedOrBeingForwarded(priorStatusWord)) {
6        // This object is forward by other threads
7        return ForwardingWord.spinAndGetForwardedObject(object, priorStatusWord);
8      } else {
9        // Forward this object
10       ObjectReference newObject = ForwardingWord.forwardObject(object, allocator
             );
11       trace.processNode(newObject);
12       return newObject;
13     }
14   } else {
15     if (testAndMark(object)) {
16       trace.processNode(object);
17     }
18     return object;
19   }
20 }
```

**Figure 3.3:** Code for mark each object

## 3.3 Linear Scan Evacuation

Since both Garbage-First and Shenandoah GC use heap linear scanning for evacuation, we implemented a linear-scan evacuation version of the simple region-based GC.

In the simple region-based GC, both object evacuation and reference updating are done together during the stop-the-world evacuation phase, using a single full heap tracing. For this linear-scan region-based GC, we divided the stop-the-world evacuation phase into two phases: The linear scan evacuation phase and the reference updating phase.

During the linear scan evacuation phase, the collector threads linear scans all the regions in the collection set and evacuate live objects in these regions. the collector does not fix or update any references during this phase.

During the reference updating phase the collector performs a full-heap tracing, just likes the evacuation phase of the simple region-based GC, but only fix and update pointers to ensure every pointer in the heap pointers to the correct copy of the object.

By separating the linear scan evacuation phase and the reference updating phase, the linear-scan region-based GC shows the basic collection processes of most G1 family of garbage collectors: `marking -> evacuation -> update references -> cleanup`. The modular design of this collectors enables the future refinements on some specific phases, e.g. concurrent marking, remembered-set based evacuation or concurrent evacuation.

## 3.4 Concurrent Marking

Another refinement we did is to make the marking phase executes mostly in concurrent to reduce the mutator latency caused by the stop-the-world object marking. This concurrent-marking region-based GC is a refinement based on the previous linear-scan region-based GC.

We use a Snapshot at the Beginning (SATB) algorithm Yuasa [1990] to complete most of the marking work concurrently without pausing mutators. The SATB algorithm assumes an object is live if it was reachable (from the roots) at the start of the concurrent marking or if it was created during the concurrent marking Yuasa [1990].

The whole marking phase consists of two pauses, initial mark pause and final mark pause. During the initial mark pause, same as the stop-the-world marking phase, the collector clears the live bitmap for all regions, and then it scans and process all the root objects including stack objects and global objects. If the mutators allocate spaces too quick and the used ratio of the heap reaches the stop-the-world GC threshold previously defined in the simple region-based GC, the collector then pauses all the mutators and switches to a stop-the-world marking phase to continue the marking process.

After the stop-the-world initial mark pause, the collector resumes the execution of all mutators and initiates the concurrent marking phase, which marks all the remaining objects in concurrent while the mutators are still in execution.

### 3.4.1 SATB write barriers

To maintain the invariant of "An object is live if it was live at the start of marking", the mutators in this concurrent-marking GC implemented a "deletion barrier", which are code fragments inserted into the java program, before every object field write, object field compare & swap and object array copy in the program. Figure 3.4 shows the object reference write barrier that is used to track object graph modifications during the concurrent marking. When a object reference field modification happens, since the old child object reference was reachable from the roots before this modification, it should be considered as a live object. So the SATB barrier traces and enqueues such old object reference fields to ensure they are live. Same as the stop-the-world marking process, the concurrent marking phase finishes after the global object queue is drained.

During the final mark pause, the collector releases and resets the marking buffer that is used for concurrent marking. Then it starts the collection set selection phase, just like the simple region-based GC.

This concurrent-marking region-based GC is an important refinement which enables the analysis of the pause time and mutator latency provided by the SATB concurrent marking algorithm on the region space. Although MMTk provides an implementation of SATB algorithm, but the old implementation is only performed

on the mark-sweep space, not on the region space. By analyzing this concurrent-marking region-based GC, it is able to understand the detailed performance impact of the Garbage-First and Shenandoah GC due to the SATB algorithm.

## 3.5 Garbage-First GC

Garbage-First (G1) GC (Detlefs et al. [2004]) was originally designed by Oracle to replace the old Concurrent Mark and Sweep GC. G1 GC was designed to target large heap but has a reasonable and predictable GC pause time. To achieve a short pause time, G1 uses a data structure called remembered-set to perform partial heap scanning during the pointer updating phase, instead of the full-heap scanning. To make the GC time predictable, a pause time prediction model is involved in G1 to predict and choose the number regions in the collection set to meet a soft real-time pause goal. In addition, G1 has a generational mode which has some minor GCs that collects young (newly allocated) regions only.

During each GC cycle of G1 GC, there are 5 major phases. The first three phases are the concurrent marking phase, the collection set selection phase and the linear scan evacuation phase, same as the concurrent-marking region-based GC. The fourth phase is the remembered-set based pointer updating phase. And the last phase is the cleanup phase which frees the regions in the collection set.

The construction of the Garbage-First GC on JikesRVM is based on the concurrent-marking region-based GC. It involves three refinements: remembered-set based pointer updating, pause time predictor and generational collection. The remembered-set based pointer updating largely reduces the pause time cause by the reference updating phase by avoiding full heap tracing when reference updating, using remembered sets. The pause time predictor uses a statistical prediction model to make the pause time of each GC more predictable and not exceeds a soft pause time goal at most of the time. The generational mode enables G1 to collect garbage more efficiently, under the help of the generation assumption of objects, which under most cases further reduces the average pause time of GCs

### 3.5.1 Remembered-set

Under the G1 collection policy, the region space further divides regions into fixed 256 B cards for constructing remembered sets.

Figure 3.5 show the general structure of a remembered-set. The remembered-set is a data structure for each region that remembers cross region pointers in other regions that pointing to objects in the current region. A remembered-set for a region is implemented as a list of PerRegionTables. A PerRegionTable in the remembered-set is a bitmap corresponds to a region in the heap. The bitmap records cards in the corresponding region that contains pointers pointing to the region that the current remembered-set corresponds to. Each bit in the bitmap corresponds to one card.

The remembered-sets is maintained by the collector and is always contains cards in other regions that has pointers to this region. The remembered-sets should be

```
@Override
protected void checkAndEnqueueReference(ObjectReference ref) {
  if (!barrierActive || ref.isNull()) return;

  if (Space.isInSpace(Regional.RS, ref)) Regional.regionSpace.traceMarkObject(
      remset, ref);
  else if (Space.isInSpace(Regional.IMMORTAL, ref)) Regional.immortalSpace.
      traceObject(remset, ref);
  else if (Space.isInSpace(Regional.LOS, ref)) Regional.loSpace.traceObject(
      remset, ref);
  else if (Space.isInSpace(Regional.NON_MOVING, ref)) Regional.nonMovingSpace.
      traceObject(remset, ref);
  else if (Space.isInSpace(Regional.SMALL_CODE, ref)) Regional.smallCodeSpace.
      traceObject(remset, ref);
  else if (Space.isInSpace(Regional.LARGE_CODE, ref)) Regional.largeCodeSpace.
      traceObject(remset, ref);
}
```

(a) The SATB Barrier which enqueues objects into a SATB buffer

```
@Inline
public void objectReferenceWrite(ObjectReference src, Address slot,
    ObjectReference tgt, Word metaDataA, Word metaDataB, int mode) {
  if (barrierActive) checkAndEnqueueReference(slot.loadObjectReference());
  VM.barriers.objectReferenceWrite(src, tgt, metaDataA, metaDataB, mode);
}

@Inline
public boolean objectReferenceTryCompareAndSwap(ObjectReference src, Address
    slot, ObjectReference old, ObjectReference tgt, Word metaDataA, Word
    metaDataB, int mode) {
  boolean result = VM.barriers.objectReferenceTryCompareAndSwap(src, old, tgt,
      metaDataA, metaDataB, mode);
  if (barrierActive) checkAndEnqueueReference(old);
  return result;
}

@Inline
public boolean objectReferenceBulkCopy(ObjectReference src, Offset srcOffset,
    ObjectReference dst, Offset dstOffset, int bytes) {
  Address cursor = dst.toAddress().plus(dstOffset);
  Address limit = cursor.plus(bytes);
  while (cursor.LT(limit)) {
    ObjectReference ref = cursor.loadObjectReference();
    if (barrierActive) checkAndEnqueueReference(ref);
    cursor = cursor.plus(BYTES_IN_ADDRESS);
  }
  return false;
}
```

(b) SATB mutator barriers

**Figure 3.4:** Snapshot at the Beginning Barriers

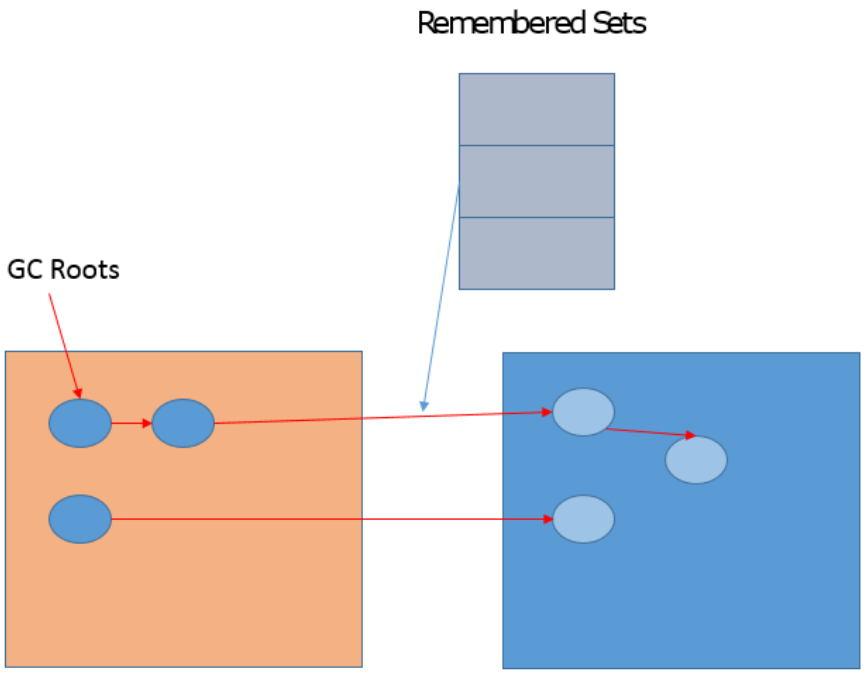


**Figure 3.5:** Remembered-Set Structure

updated at every object field modification bu mutators and after the evacuation of each object in the collection set to meet the correctness of the remembered-sets.

### 3.5.2 Concurrent remset refinements

As shown in figure 3.6, in order to maintain the structure of remembered-sets, a new barrier called "remembered-set barrier" was involved for each object field modification action in the java program. For each object field modification `obj.x = y`, the remembered-set barrier checks whether the pointer `y` is a cross region pointer (i.e. not pointing to the region that contains `obj`). If the check succeed, the remembered-set barrier enters a slow path which marks the card containing `obj` and push this card into a local `dirtyCardBuffer`. Since the minimum memory allocation unit in the MMTk metadata space is one page (4 KB), the size of the local `dirtyCardBuffer` is 1024 cards, instead of 256 cards in the original G1 implementation in OpenJDK Detlefs et al. [2004].

When the local `dirtyCardBuffer` is full, the remembered-set barrier pushes the local `dirtyCardBuffer` to the global filled RS buffer set. And when the size of the global filled RS buffer set reaches a threshold of 5 `dirtyCardBuffers`, a concurrent remset refinement is triggered. A separate concurrent remset refinement thread was started to process each `dirtyCardBuffer` in the global filled RS buffer set. The refinement thread scans each card for each `dirtyCardBuffer`. If the card is marked, clear its

```
1  @Inline
2  void markAndEnqueueCard(Address card) {
3    if (CardTable.attemptToMarkCard(card, true)) {
4      remSetLogBuffer().plus(remSetLogBufferCursor << Constants.
           LOG_BYTES_IN_ADDRESS).store(card);
5      remSetLogBufferCursor += 1;
6      if (remSetLogBufferCursor >= REMSET_LOG_BUFFER_SIZE) {
7        enqueueCurrentRSBuffer(true);
8      }
9    }
10 }
11
12 @Inline
13 void checkCrossRegionPointer(ObjectReference src, Address slot, ObjectReference
       ref) {
14   Word x = VM.objectModel.objectStartRef(src).toWord();
15   Word y = VM.objectModel.objectStartRef(ref).toWord();
16   Word tmp = x.xor(y).rshl(Region.LOG_BYTES_IN_REGION);
17   if (!tmp.isZero() && Space.isInSpace(G1.G1, ref)) {
18     Address card = Region.Card.of(src);
19     markAndEnqueueCard(card);
20   }
21 }
```

**Figure 3.6:** Remembered-set Barrier

marking data, linear scan it for cross region pointers and updates the corresponding remembered-set for each cross region pointer.

### The card table

To perform card marking and card linear scanning during concurrent remset refinements, A card table should be used to record the marking data for each card as well as the offset for the first and the last object of each card. The card table consists of three parts. a) A bitmap of all cards in the java heap to record the marking data of the cards. b) A byte array (i.e. a byte map) to record the offset of the first object for each card. c) Another byte array to record the offset of the last object for each card.

### Hot cards optimization

During concurrent remset refinements, some cards may be enqueued and scanned multiple times which brings extra computation costs. To avoid such redundant scanning, we assign a hotness value for each card. Every scan on a card increases its hotness value by 1. When the hotness for a card exceeds a hotness threshold (default is 4), this card is considered as a "hot card". Hot cards are pushed into a hot cards queue and the processing of all the hot cards card are delayed until the start of the evacuation phase.

### 3.5.3 Evacuation phase

G1 uses linear scan evacuation, just like the previous discussed collectors. During the evacuation phase, the collector linear scans each region in the collection set and copy live objects to the to-regions.

### 3.5.4 Remembered set based pointer updating

By performing concurrent remset refinements, the structure of all remembered-sets are always maintained correctly. Which enables the partial evacuation based on remembered sets without a full heap tracing.

After the collection set selection phase, the collector firstly performs a linear scan over regions in the collection set to evacuate all love objects in the collection set. Then the collector starts a references updating phase which considers the root set as all root objects plus objects in the cards that is recorded in the remembered-sets. During references updating phase, instead of performing a full heap tracing to fix and update all the references in the heap, the collector only scans root objects and cards in remembered-sets of regions in the collection-set to update references, since all pointers that needs to be updated are either root pointers or are remembered in the remembered-sets.

At the end of the pointer updating phase, same as the full-heap tracing based pointer updating, the collector frees all the regions in the collection set.

By the remembered set based pointer updating, the G1 collector has the ability to collect any subset of regions in the heap, without scanning the whole heap, as long as the collection set size is not greater than the free memory size in the heap. In this way the GC pause time due to object evacuation can be largely shortened.

### 3.5.5 Pause time predictor

Due to the ability of collecting any subset of regions in the heap, G1 can further make the pause time of each GC predictable by choosing the number of regions to in the collection set.

The total kinds of works during the stop-the-word evacuation is fixed: dirty cards refinements, object evacuation and linear scan cards for updating references. This brings us the ability to model the pause time for each GC, in terms of the remembered-set size and live bytes of each region in the collection set.

Our implementation reuses the pause time prediction model that was proposed in the original G1 paper (Detlefs et al. [2004]).

$$T_{CS} = T_{fixed} + T_{card} * N_{dirtyCard} + \sum_{r \in CS} (T_{rs\ card} * rsSize(r) + T_{copy} * liveBytes(r))$$

Where
$T_{CS}$ is the total predicted pause time
$T_{fixed}$ is the time of all extra works

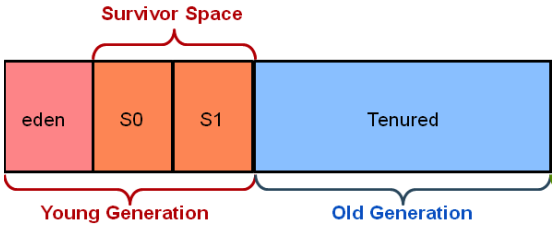


**Figure 3.7:** Generational G1 Heap Allocation

$T_{card}$ is the time of linear scanning a card for remembered set refinements
$N_{dirtyCard}$ is the number of dirty cards that has to be processed before evacuation
$T_{rs\ card}$ is the time of linear scanning a card in the remembered-set for evacuation
$rsSize(r)$ is the number of cards in the remembered-set of region $r$
$T_{copy}$ is the time for evacuating a byte
$liveBytes(r)$ is the total live bytes for evacuation

By using this pause time prediction model, during each collection set selection phase, the collector can choose the number of regions in the collection set to meet a user-defined pause time goal. This mechanism makes the pause time more predictable and being controlled around a reasonable value.

### 3.5.6 Generational G1

The original design of G1 GC comes with a generational mode which collects newly allocated regions only during young GCs (Detlefs et al. [2004]). The generational collection is based on an assumption that the newly allocated objects has higher chances to become garbage than those objects that is survived after several GCs.

Based on this assumption on the age of objects, we divided the regions into three generations: Eden, Survivor and Old generation, as shown in figure 3.7. Eden regions contain objects that is newly allocated since last GC. Survivor regions contain objects that is survived from last GC. And Old regions contain objects that is survived after several GCs. During the allocation process, the newly allocated regions are marked as Eden regions. When the ratio of the number of Eden regions exceeds a `newSizeRatio` threshold, a young GC is triggered which only collects all Eden and Survivor regions. During young GCs, live objects in Eden regions are evacuated to Survivor regions and objects in Survivor regions are evacuated to Old regions. Objects evacuated to Survivor regions will still be included into the collection set during the next young GC.

#### Young GC

Young GCs are just simple nursery GCs that only evacuate objects and not marking them. There is no marking phase during a young GC. Instead of determine the liveness of objects by marking, during young GCs the collector considers all objects

that are not in the to-space (i.e. the collection set) as live. The collector simply starts from the the root objects and the remembered-set to recursively evacuate live objects out of the collection-set.

#### Mixed GC

When the allocate memory in the heap exceeds some threshold, the generational G1 will initiate a concurrent marking phase for a mixed GC just likes the non-generational G1. When the to-space is exhausted during mixed GCs, G1 switches to a fully stop-the-world full GC.

#### Pause time predictor for young GCs

To meet the soft pause time goal during young GCs, the collector update the value of `newSizeRatio` at the end of every young GC to find a more appropriate Eden size ratio so that the collector can performs better in meeting the pause time goal during future young GCs.

## 3.6 Shenandoah GC

Shenandoah GC is a experimental garbage collector and is originally implemented on OpenJDK. Shenandoah GC is designed to reduce GC pause times for large heaps and tries to make the GC pause time not proportional to the heap size.

When the heap occupancy reaches a threshold ratio (default is 20%), the Shenandoah GC triggers a concurrent GC cycle, starting from a concurrent marking phase. The concurrent marking phase also uses the Snapshot at the Beginning algorithm, which is similar to the G1 GC and the concurrent-marking region-based GC. After the concurrent marking phase is a concurrent remembered-set selection phase which is the same as the stop-the-world remembered-set selection phase but runs in concurrent, without pausing the mutators. The third phase is the concurrent evacuation phase. Shenandoah GC uses the Brooks-style indirection pointer (Flood et al. [2016]) to evacuate objects in to-space in concurrent. The fourth phase is the concurrent reference updating phase, which also performs a full heap tracing, like the stop-the-world reference updating phase, but runs in concurrent, under the help of the Brooks-style indirection pointers.

By making the marking, evacuation and pointer updating phases run in concurrent, Shenandoah GC performs most of the heap scanning works in concurrent, without pausing mutators. In this way Shenandoah GC can have very low pause times and the pause time is not proportional to the heap size.

### 3.6.1 Brooks indirection barrier

The Brooks-style indirection pointer is a pointer stored in the java object header to record the forwarding status of a java object (Flood et al. [2016]). This requires the

```
1  // class ShenandoahMutator
2
3  @Inline
4  public ObjectReference getForwardingPointer(ObjectReference object) {
5    return ForwardingWord.extractForwardingPointer(object);
6  }
7
8  @Inline
9  public ObjectReference objectReferenceRead(ObjectReference src, Address slot,
       Word metaDataA, Word metaDataB, int mode) {
10   return VM.barriers.objectReferenceRead(getForwardingPointer(src), metaDataA,
         metaDataB, mode);
11 }
12
13 // class ForwardingWord
14
15 @Inline
16 public static ObjectReference extractForwardingPointer(ObjectReference oldObject
       ) {
17   return oldObject.toAddress().loadWord(FORWARDING_POINTER_OFFSET).and(
         FORWARDING_MASK.not()).toAddress().toObjectReference();
18 }
```

**Figure 3.8:** Brooks read barrier

Shenandoah collector in JikesRVM to reserve an extra GC word for each java object. During object allocation, after a new java object is allocated, the indirection pointer of this java object is initialized to point to the object itself.

During the concurrent evacuation phase, after an object is forwarded, the collector atomically update the indirection pointer of the old copy to point to the new copy. Mutators should always perform modification actions on the new copy of the object, by following the indirection pointer in the object header.

By using the Brooks-style indirection pointers, the mutators can read and modify object (by following the indirection pointers) while the collector can also evacuate these objects in concurrent.

### Read Barriers

The read operations of every object field `obj.x`, including the object reference fields and the primitive fields, should go through the object's indirection pointer. Figure 3.8 shows the read barrier that is inserted before every object field read instruction. The barrier always unconditionally extract the indirection pointer from the object header, and reads the object field value from the object that the indirection pointer points to.

If the object is not in the collection set or is in the collection set but is not forwarded, reading data from its original copy is safe since there is no other copies of the object at the time the object field read happens. If the object is forwarded, its indirection pointer is pointed to the new copy of the object. Then by following the indirection pointer, the mutator can always read field data from the object's new copy.

#### Write Barriers

However, since the mutator should always meet the "write in the to space" invariant, it is not safe for mutators to write objects just by following the indirection pointers. If a mutator is trying to updating a object while a collector thread is also copying this object, the write operation will only performs on the old copy and the new copy may still contains the old data, because the indirection pointer is still pointing to the old copy when the evacuation of this object is in progress.

To resolve this problem, the write barrier used in Shenandoah GC is different from the read barrier. As shown in figure 3.9, before the mutator performs the object field write operation on an object that is in the collection set, the mutator firstly checks if the indirection pointer of the object is marked as `FORWARDED` or `BEING_FORWARDED`. If the object is forwarded, then the mutator follows the indirection pointer to perform the write operation on the new copy. If the object is marked as being forwarded, the mutator awaits until the object is forwarded to get the correct indirection pointer to the new copy. If the object is marked as not forwarded, the mutator should take the responsibility to forward this object. Under such situation, the mutator copies this object and updates the indirection pointer, just like the forwarding process of the collector threads.

After the mutator takes the responsibility of forwarding unforwarded from-space objects, the mutator now meets the "write in the to space" invariant.

### 3.6.2 Concurrent evacuation

The evacuation phase is done mostly in concurrent. Before concurrent evacuation, the collector firstly collects and evacuate all root objects stop-the-world. Then during concurrent evacuation, the collector linear scans each region in the collection set to evacuate all live objects and atomically set their indirection pointers.

JikesRVM has its own implementation of object forwarding functions, but it stores the forwarding pointer into the status word in the object header. But by using the Brooks-style indirection pointer, all objects should have a valid indirection pointer in the header, which will override other status information, e.g. lock and dynamic hash status. So we use our own implementation of object forwarding functions which increases the java header by one word and stores the indirection pointer in this extra word.

#### Monitor objects

Monitor objects is an exception that is handled specially in the JikesRVM implementation of the Shenandoah GC. In JikesRVM, the jvm `monitorenter` and `monitorexit` instruction does not triggers read barriers when locking objects, which can cause inconsistencies of the lock status of the object. We modified the JikesRVM's monitor lock and unlock code to make them trigger the read barriers.

```
1  @Inline
2  public ObjectReference getForwardingPointerOnWrite(ObjectReference object) {
3    if (brooksWriteBarrierActive) {
4      if (isInCollectionSet(object)) {
5        ObjectReference newObject;
6        Word priorStatusWord = ForwardingWord.attemptToForward(object);
7        if (ForwardingWord.stateIsForwardedOrBeingForwarded(priorStatusWord)) {
8          // The object is forwarded by other threads
9          newObject = ForwardingWord.spinAndGetForwardedObject(object,
               priorStatusWord);
10       } else {
11         // Forward this object before write
12         newObject = ForwardingWord.forwardObjectWithinMutatorContext(object,
               ALLOC_RS);
13       }
14       return newObject;
15     } else {
16       return object;
17     }
18   } else {
19     return getForwardingPointer(object);
20   }
21 }
22
23 @Inline
24 public void objectReferenceWrite(ObjectReference src, Address slot,
       ObjectReference tgt, Word metaDataA, Word metaDataB, int mode) {
25   ObjectReference newSrc = getForwardingPointerOnWrite(src);
26   VM.barriers.objectReferenceWrite(newSrc, tgt, metaDataA, metaDataB, mode);
27 }
```

**Figure 3.9:** Brooks write barrier

```
1  @Override
2  @Inline
3  public void processEdge(ObjectReference source, Address slot) {
4    ObjectReference oldObject, newObject;
5    do {
6      oldObject = slot.prepareObjectReference();
7      newObject = traceObject(oldObject);
8      if (oldObject.toAddress().EQ(newObject.toAddress())) return;
9    } while (!slot.attempt(oldObject, newObject));
10 }
```

**Figure 3.10:** Concurrent update references

### 3.6.3 Concurrent updating references

Shenandoah GC also performs the update references phase in concurrent. The implementation is similar to the concurrent marking phase, but in addition to concurrently marks objects again, the collector also updates the object reference fields for each objects and make them points the correct object.

Since the java program is running during the concurrent reference updating phase, unconditionally update object reference fields may cause racing problems. So instead of unconditionally updating pointers we used in the previous stop-the-world GCs, we performs atomic pointer updating, as shown in figure 3.10, to avoid racing problems.

#### Object comparisons

Since the mutator can either load a old or a new object reference of a same object from some object fields, simply comparing addresses of objects for the `if_acmpeq` and `if_acmpne` instruction can cause false negatives. We implemented a new object comparison barrier, as shown in figure 3.11, to compare object references. In stead of simply comparing the addresses of the objects, the object comparison barrier also compares the indirection pointers of the objects.

The comparison of the original object addresses is still required. If we only compare the indirection pointers $a' == b'$, the GC can update the indirection pointer $b'$ to a new address after the barrier extracted $a'$ and before extracting $b'$, which can still cause false negatives if $a$ and $b$ is the same object.

The object comparison barrier is implemented for both JikesRVM's baseline and optimizing compiler.

#### Object compare and swaps

When the java mutator is performing object reference compare and swap operations, the collector threads can also updating the pointers contained by the slot that the CAS is operating on. Under such situation the CAS may failed even when there is only one mutator thread updating the slot, since the old value hold by the object field slot is updated by the collector in concurrent.

```
1 @Inline
2 public boolean objectReferenceCompare(ObjectReference lhs, ObjectReference rhs)
      {
3   if (lhs.toAddress().NE(rhs.toAddress()) && getForwardingPointer(lhs).toAddress
        ().NE(getForwardingPointer(rhs).toAddress())) {
4     return false;
5   } else {
6     return true;
7   }
8 }
```

**Figure 3.11:** Object Comparison Barrier

```
1 @Inline
2 public boolean objectReferenceTryCompareAndSwap(ObjectReference src, Address
      slot, ObjectReference old, ObjectReference tgt, Word metaDataA, Word
      metaDataB, int mode) {
3   boolean result = VM.barriers.objectReferenceTryCompareAndSwap(src, old, tgt,
        metaDataA, metaDataB, mode);
4   if (!result) {
5     result = VM.barriers.objectReferenceTryCompareAndSwap(src,
          getForwardingPointer(old), tgt, metaDataA, metaDataB, mode);
6   }
7   return result;
8 }
```

**Figure 3.12:** Object Reference Compare and Swap Barrier

Figure 3.12 shows the object compare and swap barrier used in Shenandoah GC. Instead of simply compare and swap on the old object, after the CAS on the old object failed, we also performs another CAS operation where the old value is the indirection pointer of the old object.

### 3.6.4 Concurrent cleanup

The last phase of Shenandoah GC is the concurrent cleanup phase, during which the collector concurrently visits each region in the collection set, clear their metadata and release these regions. This phase runs in concurrent without pausing mutators.

## 3.7 Summary

Summary what you discussed in this chapter, and mention the story in next chapter. Readers should roughly understand what your thesis takes

Chapter 4

# Performance Evaluation

This chapter discusses the performance evaluation we undertaken for analyzing the Garbage-First family of garbage collectors. This chapter will firstly discusses the software and hardware platform we used for benchmarking, Then the detailed evaluation process and resutls of both pause time and barrier latency will be presented and discussed.

## 4.1 The Dacapo Benchmark

We performed all the following pause time evacuations and barrier latency evaluation by using the Dacapo Benchmark. The benchmarking suites we

## 4.2 Hardware Platform

During the implementation of all the G1 family of garbage collectors in chapter 4.2, a list of machine with a large varities on CPU types, clock, number of processors and the size of cache and memory were involved, as shown in Table **??**. By executing all the benchmarking suite of the Dacapo Benchmark on these different machines, and thanks to the benchmarking suites of the Dacapo Benchmark which reflect that different categories of programs in the real world, we has the ability to statistically verify the correctness of the prevously implemented garbage collectors (in chapter ) and make sure they performs as intended in a real world setting.

For further performance evaluation, we chose the "fisher" machine for benchmarking.

| Machine | bobcat | rat | fisher | ox |
|---|---|---|---|---|
| **CPU Info** | AMD FX-8320 | Intel i7-4770 | Intel i7-6700k | Intel Xeon Gold 5118 |
| **Clock** | 3.5GHz | 3.4GHz | 4GHz | 2.3GHz |
| **Processors** | 4/8 | 4/8 | 4/8 | 4/48/96 |
| **L2 Cache** | 4MB | 4MB | 1MB | 48MB |
| **RAM** | 8GB | 8GB | 16GB | 512GB |

**Table 4.1:** Machines used for development and evaluation.

## 4.3 Pause Time Evaluation

### 4.3.1 MMTk harness callbacks

### 4.3.2 Mutator latency timer

### 4.3.3 Results

### 4.3.4 Discussion

## 4.4 Barrier Latency Evaluation

### 4.4.1 Methodology

### 4.4.2 Snapshot-at-the-begining barriers

### 4.4.3 Remembered set barriers

### 4.4.4 Brooks indirection pointer barriers

### 4.4.5 Discussion

## 4.5 Summary

Chapter 5

# Future Work

This chapter will discusses the future work we plan to do after the completion of thie thesis. It mainly includes: 1. Resolve a data rece problem for Shenandoah GC. 2. Perform some optimizations 3. Start wkring on ZGC.

## 5.1 Race Problem for Shenandoah GC

Currently our implementation of Shenandoah GC has a data race problem. Based on our previous analysis, the mutator is not lock and release a monitor expectedly during the concurrent phase of the Shenandoah GC, when there can be two different copies of an object exist in the heap.

Duo to the time scope of thie thesis, we cannot solve this issue currently. But we have implemented a work-around in the Shenandoah GC, with lower performance. As the most urgent problem we are facing, we plan to solve this issue at first after this project.

## 5.2 Optimizations

We are still considering the possibility to make the implemented Garbage-first and Shenandoah GC become production ready. One major part missing is the Optimizations since the development of these garbage collectors was following the original design of these collectors and did not have too much optimizations. After performing some optimizations on these collectors as well as some additional correctness verfication, the collector can have the possibility to become production ready.

## 5.3 ZGC

Due to the time scope of this project, we did not implement and measure ZGC. In addition, since JikesRVM and MMTk only supports the 32bit address space but the pointer coloring process in ZGC requires the 64bit address space, this makes the implementation of ZGC more difficult. But as a latest member of the Garbage-first family of collectors, ZGC is worth for implementation and measurements. However,

a detailed plan of resolving several hardware incompactabilities should be done in the future, before starting the implementation of ZGC.

## 5.4 Summary

This chapter will discusses the future work we plan to do after the completion of thie thesis. The next chapter contains the conclusion we obtained from the development and the analysis results of the Garbage-first family of garbage collectors.

Chapter 6

# Conclusion

Summary your thesis and discuss what you are going to do in the future in Section 6.1.

## 6.1 Future Work

Good luck.

# Bibliography


Blackburn, S. M.; Cheng, P.; and McKinley, K. S., 2004. Oil and water? high performance garbage collection in java with mmtk. In *Software Engineering, 2004. ICSE 2004. Proceedings. 26th International Conference on*, 137–146. IEEE. (cited on page 2)

BrianDemsky, K. L. G., 2015. Speculative region-based memory management for big data systems. (2015). (cited on page 1)

Detlefs, D.; Flood, C.; Heller, S.; and Printezis, T., 2004. Garbage-first garbage collection. In *Proceedings of the 4th international symposium on Memory management*, 37–48. ACM. (cited on pages 8, 9, 18, 20, 22, and 23)

Detlefs, D. L.; Martin, P. A.; Moir, M.; and Steele Jr, G. L., 2002. Lock-free reference counting. *Distributed Computing*, 15, 4 (2002), 255–271. (cited on page 6)

Endo, T.; Taura, K.; and Yonezawa, A., 1997. A scalable mark-sweep garbage collector on large-scale shared-memory machines. In *Proceedings of the 1997 ACM/IEEE conference on Supercomputing*, 1–14. ACM. (cited on page 7)

Flood, C. H.; Kennke, R.; Dinn, A.; Haley, A.; and Westrelin, R., 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. ACM. (cited on page 24)

Lieberman, H. and Hewitt, C., 1983. A real-time garbage collector based on the lifetimes of objects. *Communications of the ACM*, 26, 6 (Jun. 1983), 419–429. doi: 10.1145/358141.358147. (cited on page 9)

Lins, R. D., 1992. Cyclic reference counting with lazy mark-scan. *Information Processing Letters*, 44, 4 (1992), 215–220. (cited on page 7)

Moon, D. A., 1984. Garbage collection in a large LISP system. In LFP '84: *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming* (Austin, Texas, USA, Aug. 1984), 235–246. ACM, New York, New York, USA. doi:10.1145/800055.802040. (cited on page 9)

Ungar, D., 1984. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. In SDE 1: *Proceedings of the 1st ACM SIGSOFT/SIGPLAN*

*Software Engineering Symposium on Practical Software Development Environments* (Pittsburgh, Pennsylvania, USA, Apr. 1984), 157–167. ACM, New York, New York, USA. doi:10.1145/800020.808261. (cited on page 9)

Yuasa, T., 1990. Real-time garbage collection on general-purpose machines. *Journal of Systems and Software*, 11, 3 (1990), 181–198. (cited on pages 12 and 17)