Understanding & Analyzing the Garbage-first Family of Garbage Collectors

Wenyu Zhao

A thesis submitted for the degree of
Bachelor of Science (Honours) at
The Research School of Computer Science
Australian National University

October 2018

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

Wenyu Zhao
25 October 2018

To my partner and my parents.

Acknowledgments

First, I would like to thank my supervisor, Prof. Steve Blackburn. When I first came to Steve's lab, he did what he can to help me get familiar with this project and the area of virtual machines and garbage collection. Steve also gave me a lot of instructions with his wisdom and knowledge and helped me solved all the problems I was facing throughout my research project this year. Steve's knowledge and experience in the area of memory management and programming language impressed me a lot. Having academic discussions and debates with Steve was a great fun and I would love to continue working with him in the future if possible. In addition, Steve provided me with a lot of opportunities in the area of research, like funding me for attending conferences, which strongly helped me to become a better researcher.

I would like to thank other members in our research lab, especially Branda Wang, Zixian Cai and Pavel Zakopaylo, who gave me much of help with the project, especially when I first came to this lab and barely know nothing about garbage collection and Java Virtual Machines. Also, I want to thank Lin Yi and Xi Yang, although we have never met, your useful instructions and advice about the benchmarks made my life much easier when I was performing measurements for my project.

Also, I want to thank my partner Tianjia Zhao who gave me plenty of support throughout this year. Conducting a one-year research project alongside with four hard computer science courses is no easy, and you always try your best to comfort me every time when I was under stress.

Finally, I would like to thank my parents Houjian Zhao and Ying Qiang, for raising me and provided me with a good environment for my development. Especially, you inspired my interest in science when I was a young child, which I will never forget in the rest of my life. Although we are in separate countries this year, you always care about my life and provide me with plenty of support for my life and research.

Abstract

Region-based garbage collectors, including Garbage-first GC, Shenandoah GC, C4 GC, and ZGC, share a lot of similar designs and algorithms, such as the similar marking algorithm, memory structure, and allocation algorithm. However, the existence of such underlying relationship among these collectors has not been well identified and explored. Instead, the designers of these region-based collectors tend to treat these as stand-alone collectors instead of an improvement over existing collectors. Such ignorance of the underlying relationships among the region-based collectors can mislead the future design of related garbage collectors.

Hence, analysis, measurements, and comparisons among these collectors can also be hard. Although the design and algorithms of these collectors are similar, their structural relationships are not reflected in the original design and implementations of these collectors. For this reason, no one has measured GC performance contribution of specific parts of the GC algorithm or the extension component involved in these collectors. Which means no one can properly understand the pros and cons of the design of these collectors, and may further lead to some potential performance issues due to the inappropriate GC design.

The key contribution of this thesis is the identification of the existence of a structural relationship among the region-based collectors and an in-depth exploration of the structural relationships among a set of region-based collectors which have a similar design to the Garbage-first GC. In this thesis, I use the term "The Garbage-first Family of Garbage Collectors" to describe such category of collectors.

This thesis produces the first implementation to reflect these structural relationships. Specifically, in this thesis, I built a total of six members of the G1 family of collectors, starting from a simple region-based collector to the Garbage-first GC and Shenandoah GC. Each collector is implemented as an improved versions of the previous collector to reflect the corresponding algorithmic relationship.

Based on such implementation, this thesis performs a detailed analysis of GC performance contribution of each component of the algorithms. These analyses include the measurement of the GC pause time, concurrency overheads of several algorithms and the space overhead of remembered-sets.

The exploration of the Garbage-first family of garbage collectors leads to the conclusion that there exists a structural relationship among these G1 family of collectors. Instead of being stand-alone collectors, they tend to be collectors with algorithmic improvements over existing collectors.

As the result of GC performance evaluation, structural components including concurrent-marking, remembered-sets, and concurrent evacuation contributes to a reduction of 58.2%, 41.9%, and 84.1% respectively to the 95 percentile GC pause time on the DaCapo benchmark suite. By using remembered-sets, G1 has 9.6% average footprint overhead. G1 has a concurrency overhead of 22.0% due to concurrent marking and an overhead of 59.6% due to concurrent remembered-set refinements. Shenandoah GC has a concurrent evacuation overhead of 85.5%.

The explored algorithmic relationships among the G1 family of collectors can help GC designers or other programmers working with the JVM to have a deeper understanding of the G1 family of collectors as well as their relationships and the pros and cons of their involved improvements. The measured GC performance contribution of each component of the GC algorithm can help garbage collection algorithm designers to reconsider the design of the region-based garbage collectors and memory structures, identify the main advantages and drawbacks of each involved GC algorithm and hence have the ability to make further optimizations to them.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This thesis explores the Garbage-first family of garbage collectors, including the underlying relationship among them, the improvements each GC makes and the performance contribution of each algorithmic component. My thesis is that different members in Garbage-first family of collectors are strongly related and that each collector is an improvement over other existing collectors. Each improvement increases GC performance in some respect but also has drawbacks.

1.1 Project Statement

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

Among all the existing low-latency garbage collectors, the category of region-based collectors is widely explored and used in industry. Region-based collectors are designed to reach high GC performance (especially GC pause time) by managing the heap as a set of memory regions. Four well-known region-based garbage collectors, Garbage-first GC, Shenandoah GC, C4 GC and ZGC created by Detlefs et al. [2004]; Flood et al. [2016]; Tene et al. [2011]; Liden and Karlsson [2018] respectively are high performance region-based garbage collectors implemented for Java Virtual Machines to achieve a short GC pause time and a high program throughput.

The design and implementation of such region-based collectors has been well explored, but unfortunately, their underlying relationship remains unidentified and unexplored. Although these collectors share a lot of basic algorithms and structures, the original papers and other publications describing these collectors only regard them as newly created and stand-alone collectors instead of improvements over the existing collectors. Such ignorance of the potential relationships prevents GC designers to have a clear overview of the algorithms supporting these collectors.

The lack of explorations of these relationships makes GC performance evaluation and analysis among these collectors difficult. The collectors are invented by different groups. The implementations and optimizations employed by these collectors are varied. This means that the original implementations of the family of collectors do not reflect their underlying structural relationships. This implementation differences make it hard to understood GC performance contribution of each GC algorithm involved in these collectors. Also, the lack of explorations on these relationships and the lack of performance analysis and comparisons may cause some unexpected performance issue to these collectors. As an example, according to Nguyen et al. [2015], sometimes GC time of region-based collectors can take up to 50% of the total execution time of Big Data systems. On the other hand, this can also mislead the future design of related collectors.

So here comes the problem which this thesis is aimed to solve: What are the relationships among the G1 family of collectors and in which way do they contribute to GC performance?

1.2 Contribution

I show that this family of collectors be strongly related to each other, although the original papers of these collectors present them each as a stand-alone collector instead of an improved collector based on other existing GC algorithms. The lack of explorations of such structural relationships makes it hard to carefully analyze the performance contribution caused by each part of the GC algorithm, hence prevents GC designers from truly understanding the pros and cons of each GC algorithm.

This thesis makes an exploration of the underlying relationship among the G1 family of garbage collectors. The algorithmic relationships are summarized as a series of progressive improvements, which generally reflects the evolutionary history of the G1 family of collectors.

As another major component of the exploration, I implement the collectors by following the steps of the progressive improvements, starting from a most simple form of the region-based collector to the most complex collector, e.g. Garbage-first GC and Shenandoah GC, to reveal the hierarchy of the G1 family of collectors. I did not implement C4 GC and ZGC due to some platform limitations, which will be discussed in Chapter 5. GC performance of these implemented collectors is also evaluated and discussed.

The general steps involved in this thesis for exploring these collectors include:

Discover the relationships of the G1 family of garbage collectors including understanding the basic algorithms of these collectors, inferring the underlying relationships among them and reconsidering them as a series of algorithmic improvements instead of individual isolated collectors.

Implement a simple mark-region collector by starting from an existing SemiSpace GC in MMTk and replacing the two copy-spaces with a region-space which divides the memory up into multiple regions. Then I added an extra marking phase before the evacuation phase.

Perform a series of improvements by starting from the simple mark-region collector and progressively perform improvements including linear-scan evacuation, concurrent marking, remembered sets and concurrent evacuation. This step produces improved versions of the mark-region collectors, the Garbage-first GC and the Shenandoah GC.

Measure GC performance including measuring the GC pause times of all implemented collectors, the concurrency overheads and the remembered set size. These measurements reflect the performance gain due to each algorithmic improvement.

Explain and explore the results which leads to a reconsideration of the pros and cons of each G1-related GC algorithm in depth.

A total of six 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.

In order to make a detailed and careful analysis of these collectors and be able to compare these collectors at an algorithmic level, the implementations are as close as possible to the design of their original papers.

I measure the GC pause times for each implemented collector, as well as the concurrency overheads involved in these collectors, including overhead for concurrent marking, concurrent remembered-set refinements and concurrent evacuation. I found that concurrent marking contributes to a decrease of the 95 percentile GC pause time by 58.2% where linear scanning evacuation increases the GC pause time by 15.8%. However, the use of remembered-sets can reduce the GC pause time by 41.9%, and concurrent evacuation can also reduce the GC pause time up to 84.1% As the result of concurrency overhead analysis, I found that concurrent marking has an overhead of 22.0% where Shenandoah GC's concurrent evacuation has an overhead of 85.5%.

All works were done in JikesRVM, a research purpose Java Virtual Machine and MMTk which is a memory management took written in Java [Blackburn et al., 2004].

1.3 Thesis Outline

Chapter 2 provides a general background and an overview of the Garbage-first family of garbage collectors. Similarities and differences among some major categories of garbage collectors are discussed as well. Also, the related work on implementing and measuring region-based collector and G1 family of collectors are discussed.

Chapter 3 provides the detailed steps and algorithms of the implementation of all the Garbage-first family of garbage collectors.

Chapter 4 describes the methodology used for evaluating the implemented collectors, including the benchmark involved and the detailed steps of each evaluation. This chapter also presents the results of the evaluation and benchmarking on the implemented collectors, as well as detailed and critical discussion on the evaluation results of the implemented collectors.

Chapter 5 discusses the work related to this project that I plan to do in the future and the conclusion of this thesis.

Chapter 2

Background and Related Work

Garbage collection is a hot topic in terms of modern managed programming language implementations. Specifically, region-based collectors like Garbage-first GC and C4 GC are widely used in modern Java Virtual Machines to archive high GC performance. This chapter describes the background and basic ideas of several garbage collectors, particularly those targeting the Java Virtual Machines and are implemented in OpenJDK, as well as the differences among them. In addition, this chapter also performs a general discussion of the related work on implementing and analyzing region-based garbage collectors.

Section 2.2 roughly discusses and compares the different classes of GC algorithms. Section 2.4 describes the general design of the Garbage-first family of garbage collectors. Section 2.5 describes the related work on implementing and analyzing region-based garbage collectors.

2.1 JikesRVM and MMTk

The whole project discussed in this thesis is based on JikesRVM. All the garbage collectors I implemented and evaluated in this project are all implemented by using the Memory Management Toolkit (MMTk). In this section, I will briefly discuss the design of JikesRVM and MMTk, as well as some introductions of their general structures.

2.1.1 JikesRVM

JikesRVM is a research Java Virtual Machine and was first released by IBM [Alpern et al., 2005]. 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.

Instead 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 bytecode into machine code and does no optimizations while the optimizing compiler performs several optimizations during the code generation phase. I performed all benchmarking and analysis works on the optimized build of the JikesRVM.

2.1.2 MMTk

MMTk is a memory management toolkit and is used as a memory management module of JikesRVM for memory allocation and garbage collection [Blackburn et al., 2004].

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 receiving an 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 have two major phases: the marking phase and the evacuation 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, I extended this to perform a separate 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 within each space allocation slow path. This involves the invocation of methods `collectionRequired(...)` and `concurrentCollectionRequired(...)`. I made full use of these two methods, not only for checking whether a collection is required but also performing switches between different schedules of collection phases for either nursery or mature collection for the G1 collector.

2.2 Categories of GC Algorithms

This section discusses the major 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. This algorithm was firstly introduced by Collins [1960]. The reference count for an object is increased when an 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 each object is initialized to one when the object is created, which means there is only one owner for the object (its creator). 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 will be swept.

This collection algorithm has two variants. The naive reference counting algorithm performs reference counting manipulations without pausing all mutators and the workload for collecting objects is almost evenly distributed. However, manipulating reference counts all the time significantly increases the mutator overhead. Another variant is the deferred reference counting, which performs object collection during some small pauses instead of taking up the mutator time. Deferred reference counting collectors generally have relatively larger pause time than naive reference counting collectors but have a much better mutator performance.

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 are no objects referencing to them.

2.2.2 Mark & sweep GC

Mark and Sweep GC was firstly invented by McCarthy [1960]. It is a type of tracing GC which uses the object graph to assist with garbage collection. The algorithm considers all the objects that are unreachable in the program as garbage. 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 mutator does 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 first scans and marks all of the static objects, 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 definitely reachable from the "root set" and all other objects become floating garbage and are swept.

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-scale servers and high memory-load programs for business become more and more popular, 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 lead 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 evacuation, which moves objects in the heap together.

As one of the most simple copying GC, the SemiSpace GC created by Fenichel and Yochelson [1969] divides the whole heap into two spaces: the from-space and the to-space. All objects allocation are done from the to-space. When the to-space is exhausted, a GC cycle is triggered. The two spaces are flipped at the start of GC. The collector starts 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 swept at the end of the GC cycle.

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

2.3 Mark-Region Collectors

Instead of evacuating all the live object during each GC like the SemiSpace GC, Mark-Region collectors divide the heap up into small regions, and only evacuate or collect objects in a specific subset of regions.

As a high performance Mark-Region collector, Immix GC created by Blackburn and McKinley [2008] has a two level heap architecture, blocks and lines. It divides the heap up into blocks and further splits blocks into lines. During the marking phase, Immix GC marks the blocks as well and collects blocks that are not marked at the end of the marking phase. During each GC cycle, Immix GC performs opportunistic evacuation after the marking phase for defragmentation. The opportunistic evacuation only copy a very small proportion of objects to other regions which significantly reduces the GC pause time due to object evacuation.

2.4 Garbage-First Family of Garbage Collectors

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

2.4.1 Garbage-first GC

Garbage-first GC created by Detlefs et al. [2004] is a copying collector which was initially released in Oracle JDK 6 and was fully supported in Oracle JDK 7. G1 was designed as a server-style collector which targeting machines with multi-processors and large memories.

G1 GC divides the whole heap up into fixed 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 G1 GC starts with a concurrent marking phase. During 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 the 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 the implementation of remembered sets can be inefficient.

2.4.2 Shenandoah GC

Shenandoah GC is 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 reduce the GC latency by performing concurrent evacuation. 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 partial evacuation to reduce pause times. Instead, the Shenandoah GC performs the evacuation phase concurrently to collect all possible regions, without pausing mutators.

By performing concurrent marking and evacuation, Shenandoah GC does most of the heap scanning work concurrently. 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 barriers is much more larger than other GCs.

2.4.3 C4 GC

C4 GC is a pauseless GC algorithm created by Tene et al. [2011] and is the default collector of Azul's Zing JVM. C4 is a region-based generational collector which performs mostly concurrent marking and evacuation during nursery and mature GCs. According to the Tene et al. [2011], by performing most of the GC work concurrently, C4 GC has the ability to reduce the mutator response time down to around 10 ms when targeting 100 GB heaps.

C4 GC is similar to Shenandoah GC, where the difference is that C4 is generational and uses an enhanced version of Shenandoah GC's mutator barrier. In detail, 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, C4 significantly reduces the throughput reduction caused by the mutator barriers.

2.4.4 ZGC

ZGC is a new garbage collector introduced by Liden and Karlsson [2018] and is very similar to the Shenandoah GC and C4 GC. ZGC was developed as an experimental pauseless collector for OpenJDK. ZGC is very similar to C4 GC. Both of them use "colored pointers" as well as related mutator barriers to perform concurrent marking and evacuation. The only difference from C4 GC is that ZGC is still under experimental status and currently does not supports generational mode.

2.5 Related Work

2.5.1 Implementations and evaluations of the G1 family of collectors

The current working G1 family of garbage collectors implemented for modern Java Virtual Machines includes the Garbage-first GC, Shenandoah GC, and ZGC. All of these three major collectors are implemented in OpenJDK.

Detlefs et al. [2004] designed and evaluated the basic algorithm of the Garbage-first collector, including the original design of the pure(non-generational) and generational version of G1 GC. Based on this paper, G1 GC was first implemented and released as an alternative experimental GC for OpenJDK 7. Flood et al. [2016] designed the basic algorithm of the Shenandoah garbage collector and the first implementation of Shenandoah GC was done as an alternative experimental GC for OpenJDK 8. ZGC is a new garbage collector designed and implemented for OpenJDK and has justly released the first experimental version in OpenJDK 11.

Detlefs et al. [2004]; Flood et al. [2016] evaluated the performance of G1 GC and Shenandoah GC respectively, including the pause time and mutator barriers overhead. However, their evaluations use different benchmarking programs and are done within different hardware platforms. In addition, the implementations between these collectors vary significantly, even for parts that share similar ideas, which makes it hard to compare the benchmarking results between G1 GC and Shenandoah GC.

2.5.2 Evaluation of region-based collectors

Gay and Aiken [1998] measured the performance of memory management on a region-based heap. They divided the heap up into regions and used a safe C dialect with a conservative reference-counting collector to manage the memory. Then they measured the performance of both allocation and collection over such region-based heap, on a collection of realistic benchmarks. They found that the regional structure has advantages for memory allocation and collection, with a competitive performance compared to `malloc/free` in C programs and has low overhead when using a garbage collector. However, they only evaluated the reference counting collector, which is not a mark and copy collector that this thesis is trying to evacuate. Also, the language they were using is a safe C dialect, which is different from the Java language in many aspects.

2.5.3 Evaluation of barrier overheads

Yang et al. [2012] evaluated a wide range of different read and write barriers on JikesRVM, using the DaCapo benchmarks and the SPECjvm98 benchmarks. Also, they observed the barrier performance differences of in-order and out-of-order machines. They found that write barriers generally have overheads of 5.4% while write barriers have average overheads of 0.9%. However, these barriers evaluations did not include the evaluations of the Brooks indirection barriers. In addition, this paper measured the behavior of card marking barriers but did not measure the overhead of the remembered-set barrier which is optionally used alongside with the card marking barrier in the Garbage-first GC.

2.6 Summary

In this section, I introduced the background and basic ideas of several garbage collectors, as well as the design of the widely used copying region-based garbage collectors. I also performed a general discussion of the related work on implementing and analyzing region-based garbage collectors.

Most mark-copy garbage collectors generally have the similar design, especially for those Garbage-first related region-based collectors which share the common design of the heap structure and marking algorithm. This leads to an open question: To what extent do these region-based collectors relate to each other? To answer this question, in the next chapter I will explore the differences and relationships among these collectors and implement them on JikesRVM as a series of progressive algorithmic improvements instead of building them separately.

Chapter 3

Design and Implementation of the G1 Family of Garbage Collectors

As briefly discussed in the previous chapter, the four well-known copying region-based collectors, Garbage-first GC, Shenandoah GC, C4 GC, and ZGC share many common design parts such as concurrent marking algorithm, region-based structure and allocation algorithm. However, there are also several differences among them, especially the different techniques designed to improve GC performance. Hence their resulting GC performances are different.

As a major contribution of this thesis, this chapter explores the underlying relationships among these collectors, which the original designers of these GCs did not identify. Based on these connections, this chapter also discusses the implementation of several region-based GCs, we call "The Garbage-first family of garbage collectors", as a series of improvements inferred by the relationships.

Section 3.1 describes the general and high-level design of the G1 family of garbage collectors and explores their underlying relationships. Section 3.2 describes the implementation details of a simple region-based GC. Section 3.3 describes the implementation of the simple region-based GC that uses linear scanning for evacuation. Section 3.4 describes the implementation of the concurrent marking algorithm. 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

The general design of the implementation including reconsidering the underlying relationship among all the existing region-based garbage collectors, reordering them and implementing them to reflect those relationships.

3.1.1 Analysis of the collectors and their relationships

The four well-known region-based garbage collectors, G1 GC, Shenandoah GC, C4 GC, and ZGC share plenty of common design and algorithms. The major similarity of these collectors is that they are copying region-based collectors designed for modern Java Virtual Machines and targeting large heaps.

The same design goal leads to the similar general structure of these collectors. As described in publications by Detlefs et al. [2004], Flood et al. [2016], Tene et al. [2011] and Liden and Karlsson [2018], their similarities include:

Region-based memory structure which divides the whole heap up into smaller regions and performs copying collection on a subset of regions during each GC (such subset is called the "collection set").

Bump pointer allocator which uses an allocation algorithm that linearly allocates objects within the memory slice of a region and moves to another available region when the current region is filled.

Concurrent marking. At the beginning of each GC cycle, all of the collectors use the Snapshot-at-the-beginning algorithm (firstly introduced by Yuasa [1990]) to mark all live objects in the heap concurrently without stopping the mutators.

Perform evacuation after the marking phase, including copying live objects in the collection set to other regions before releasing the memory in the collection set.

Differences also exist among these collectors. These garbage collectors adopt different techniques to reduce the GC pause time and increase the mutator throughput. Specifically,

Garbage-first GC uses a data structure called a "remembered set" to record the cross region pointers during the execution of the Java program. Later during evacuation phase, instead of walking over the whole object graph to update references as other collectors (e.g. SemiSpace GC) did, G1 only needs to scan the remembered sets to update cross region pointers pointing to the collection set. Also, G1 has a generational mode which eagerly collects newly allocated objects during nursery GCs.

Shenandoah GC is very similar to G1, but it performs the evacuation phase concurrently, by using the Brooks indirection barrier (which was first introduced by Brooks [1984]). Although Shenandoah GC still performs full heap tracing during the evacuation phase, the GC pause time is very low and is not proportional to the heap size.

C4 GC be simply regarded as an improvement over the Shenandoah GC with a generation mode and better mutator performance. ZGC uses the "colored pointers" as well as a highly optimized read barrier to assist with the Brooks indirection barrier to further increase the mutator throughput.

ZGC is very similar to C4 GC and also performs concurrent evacuation during each GC. The major difference to C4 GC is that ZGC is still under experimental state and currently has no generational mode.

In general, these four collectors are strongly related to each other in terms of the design and algorithm structure. In addition, each collector performs their own enhancements to improve GC performance. In this way, at the algorithmic level, these collectors tend to be a set of improvements over the naive region-based collector. Hence they are more likely to form as a family of collectors instead of individual and unrelated collectors.

3.1.2 Implementation steps

As the first implementation which reflects such algorithmic hierarchy, members of the G1 family of garbage collectors are reorganized and reimplemented on JikesRVM by following different steps against the original implementations.

I started from implementing a simple region-based GC which divides the whole Java heap up into fixed sized regions. During each GC cycle it performs fully stop-the-world but parallel object marking and full-heap tracing based object evacuation. Then I made progressive improvements based on this very simple collector to further implement some G1 family of garbage collectors.

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

Then I 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. The Snapshot-at-the-beginning algorithm [Yuasa, 1990] was used to implement the concurrent marking phase.

By adding the remembered-sets to the concurrent-marking region-based GC and switching to remembered-set based evacuation, I implemented the Garbage-first collector.

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

By performing such progressive improvements, I successfully implemented a series of G1 family of garbage collectors that share as much code and design as possible among the implementations of these collectors. This enables the possibility for future algorithmic-level analysis on these garbage collectors. The other two discussed collectors, C4 GC and ZGC, are not implemented and evaluated in this project.

3.2 Simple Region-based GC

This simple region-based GC is provided as a baseline for future implementation of the G1 family of collectors. It contains the most basic structures of the Garbage-first family of garbage collectors such as the region-based heap space and bump pointer allocation algorithm. By making progressive improvements over this region-based

GC, I keep the implementation differences among the G1 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 use 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 (256 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 code for such allocation fast path. Most of the object allocation processes will only follow the fast path. After a region is filled, to allocate a new object, the mutator enters a slow path which is shown in Figure 3.1(b). In this slow path the allocator 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 in the metadata pages for region linear scanning which is lately used in some collectors. Also, an off-heap bitmap is maintained in the region's metadata pages to record the liveness data of objects in this region.

3.2.2 Stop-the-world marking

Based on the high-level design of MMTk, after the memory is exhausted, MMTk checks if a stop-the-world or concurrent GC is required for the currently 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 `reservedPercent` (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 first scans and marks all the stack and global root objects and pushes them into an object queue. Then the collector threads keep popping objects from the object queue and collecting all its object reference fields (which are child nodes of the current object node in the object graph). If these object fields were not marked previously, the collector then marks and pushes them back into the object queue. The marking process is done when all of the local and global object queues are drained, which means all objects that are reachable from the root objects have been marked by the end of the marking phase.

Instead of using the GC byte in the object header for marking which is widely used in other GCs in MMTk, a bitmap for each region is maintained to record the liveness data of the objects. 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

```java
@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

```java
@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-based allocation algorithm

```java
@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 marking each object

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 operation succeeds.

Although using such extra "liveness table" is not necessary for this region-based GC, this is a common design for Garbage-first and Shenandoah GC. So the GC byte in the object header was disabled at the very beginning to reduce the implementation difference among all the 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 set of regions for later defragmentation.

The collection set selection phase first takes a list of all allocated regions, and sorts 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 count of live bytes in the collection set is not greater than the free memory size of the heap, to prevent the to-space from being exhausted during evacuation.

At the end of the collection set selection phase, the collector marks all regions in the collection set as "`RELOCATION_REQUIRED`" 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 in highly fragmented memory slices and copying 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 aimed to copy/evacuate all live objects in the collection set to other regions.

Figure 3.3 shows the process of evacuating an object. To evacuate objects, the collector performs another full heap tracing to re-mark all the objects, just like the

```
@Inline
public ObjectReference traceEvacuateObject(TraceLocal trace, ObjectReference
    object, int allocator) {
  if (Region.relocationRequired(Region.of(object))) {
    Word priorStatusWord = ForwardingWord.attemptToForward(object);
    if (ForwardingWord.stateIsForwardedOrBeingForwarded(priorStatusWord)) {
      // This object is forward by other threads
      return ForwardingWord.spinAndGetForwardedObject(object, priorStatusWord);
    } else {
      // Forward this object
      ObjectReference newObject = ForwardingWord.forwardObject(object, allocator
          );
      trace.processNode(newObject);
      return newObject;
    }
  } else {
    if (testAndMark(object)) {
      trace.processNode(object);
    }
    return object;
  }
}
```

Figure 3.3: Code for evacuating each object

marking phase discussed before. But in addition to marking the objects, the collector also copies the objects and atomically updates the forwarding status in the object header if the object is in the collection set. Such full heap tracing ensures that all objects that were marked in the marking phase are also being scanned and remarked in the evacuation phase. This means that 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 the G1 family of garbage collectors and can have several variants (e.g. concurrent evacuation), it is necessary to verify the correctness of the evacuation process in order to assist in debugging and gain confidence in the implementation. I 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 ensures that they are either null or pointing to the valid Java objects. The collector also checks that no object node is 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 state.

3.3 Linear Scan Evacuation

Since both Garbage-first and Shenandoah GC use heap linear scanning for evacuation, I 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, by using a single full heap tracing. For this linear-scan region-based GC, I split 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 linearly scan all of 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 fixes and updates pointers to ensure that every pointer in the heap points 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 reveals 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 redesign and rewriting of some specific phases, e.g. concurrent marking, remembered-set based evacuation or concurrent evacuation.

3.4 Concurrent Marking

An improvement over the previous collector is to make the marking phase concurrent to reduce the mutator latency caused by the stop-the-world object marking.

The concurrent marking phase uses the Snapshot-at-the-beginning (SATB) algorithm. This algorithm was first introduced by 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.

The whole marking phase consists of two pauses: the initial mark pause and final mark pause. During the initial mark pause, just as within 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. After that, the collector resumes all the mutators and concurrently marks all the remaining object. 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.

3.4.1 SATB write barriers

As part of the SATB algorithm, 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 SATB barrier, which is a piece of code inserted into the Java program, before every object field write, object field compare & swap and object array copy operations in the program. Figure 3.4 shows the object reference write barrier that is used to track every object graph modifications during the concurrent marking. When an 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 this old object reference field to ensure it is marked as live. Just as within the stop-the-world marking process, the concurrent marking phase finishes after the local and global object queues are drained.

During the final mark pause, the collector releases and resets the marking buffer that was 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 enhancement which enables the analysis of the pause time and mutator latency due to the SATB concurrent marking algorithm on the region space. By analyzing this implementation of the concurrent-marking algorithm, we are 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 replaces 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 performing a 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 performs nursery 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, just as within 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 improvements over the concurrent region-based collector discussed in Section 3.4: remembered-set based pointer updating,

```
@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

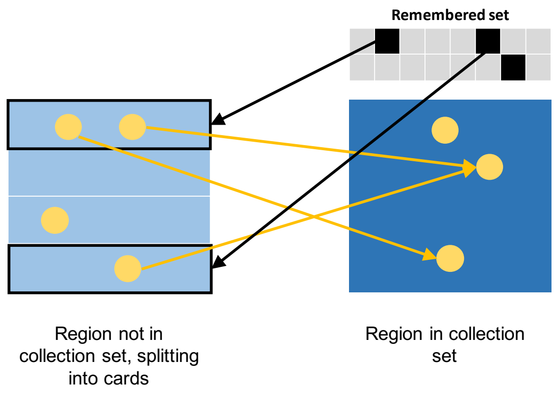

Figure 3.5: Remembered-set structure

pause time predictor and generational collection. The remembered-set based pointer updating phase largely reduces the pause time by avoiding full heap tracing when performing reference updating, by using remembered sets. The pause time predictor uses a statistical prediction model to make the pause time of each GC more predictable and most of the time not exceed a soft pause time goal. The generational mode enables G1 to collect garbage more efficiently by collecting young objects earlier.

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 shows the general structure of a remembered-set. The remembered-set [Hosking and Hudson, 1993] is a data structure for each region to remember 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 current region. Each bit in the bitmap corresponds to one card.

The remembered-sets are maintained by the collector and should be updated at every object field modification operation to ensure 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 succeeds, the remembered-set barrier enters a slow path which marks the card containing `obj` and pushes this card into a local `dirtyCardBuffer`. Since the minimum memory allocation unit in the

```
@Inline
void markAndEnqueueCard(Address card) {
  if (CardTable.attemptToMarkCard(card, true)) {
    remSetLogBuffer().plus(remSetLogBufferCursor << Constants.
        LOG_BYTES_IN_ADDRESS).store(card);
    remSetLogBufferCursor += 1;
    if (remSetLogBufferCursor >= REMSET_LOG_BUFFER_SIZE) {
      enqueueCurrentRSBuffer(true);
    }
  }
}

@Inline
void checkCrossRegionPointer(ObjectReference src, Address slot, ObjectReference
    ref) {
  Word x = VM.objectModel.objectStartRef(src).toWord();
  Word y = VM.objectModel.objectStartRef(ref).toWord();
  Word tmp = x.xor(y).rshl(Region.LOG_BYTES_IN_REGION);
  if (!tmp.isZero() && Space.isInSpace(G1.G1, ref)) {
    Address card = Region.Card.of(src);
    markAndEnqueueCard(card);
  }
}
```

Figure 3.6: Remembered-set barrier

MMTk metadata space is one page (4 KB), the size of the local `dirtyCardBuffer` is 1024 cards, instead of 256 cards which are used by the original G1 design.

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 globally filled RS buffer set. The refinement thread scans each card of each `dirtyCardBuffer`. If the card is marked, it clears its marking data, linearly scans 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 of the first and the last object for 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. This can cause extra computation costs. To avoid such redundant scanning, a hotness value is assigned for each card. Every scan on a card increases its hotness value by 1. When the hotness value for a card exceeds a threshold (default is 4), this card is considered as a "hot card". Hot cards are pushed into a separate hot cards queue and the processing of all the hot cards card is delayed until the start of the evacuation phase.

3.5.3 Evacuation phase

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

3.5.4 Remembered set-based pointer updating

By performing concurrent remset refinements, G1 GC can ensure that the structure of all remembered-sets is always maintained correctly. This enables partial heap scanning for pointer updating, without full heap tracing.

After the collection set selection phase, the collector first performs a linear scan over regions in the collection set to evacuate all live objects in the collection set. Then the collector starts a reference updating phase which considers the root set as the union of all root objects and objects in the cards that are recorded in the remembered-sets. During the reference 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 to update references, because all pointers that need to be updated are either root pointers or those that 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 performing 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. In this way, the GC pause time due to object evacuation can be largely shortened. Although G1 must still perform a full-heap tracing for marking, it performs this process concurrently instead of pausing mutators

3.5.5 Pause time predictor

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

The total works to do during the stop-the-world evacuation are 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.

My implementation reuses the pause time prediction model 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 involved during the GC pause
T_{card} is the time of linear scanning a card for remembered set refinements
$N_{dirtyCard}$ is the number of dirty cards that have to be processed before evacuation
$T_{rs\ card}$ is the time to linearly scan 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 have higher chances to become garbage compared to those objects that are survived after several GCs.

Based on this assumption on the age of objects, the Generational G1 collector divides the regions into three generations: Eden, Survivor and Old generation, as shown in Figure 3.7. Eden regions contain objects that are newly allocated since the end of last GC. Survivor regions contain objects that are survived from the last GC. And Old regions contain objects that are 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 as part of the collection set during the next young GC.

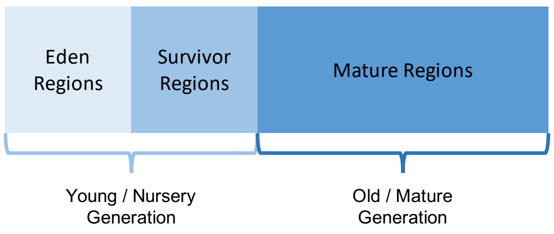

Figure 3.7: Generational G1 heap structure

Young GC

Young GCs are just simple nursery GCs that only evacuate objects without a marking phase. There is no marking phase during a young GC. Instead of determining 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 root objects and remembered-sets to recursively evacuate live objects out of the collection-set.

Mixed GC

When the allocated 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 stop-the-world full GC.

Pause time predictor for young GCs

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

3.6 Shenandoah GC

Shenandoah GC is an experimental garbage collector and is originally implemented on OpenJDK [Flood et al., 2016]. 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 concurrently, without pausing the mutators. The third phase is the concurrent evacuation phase. Shenandoah GC uses the Brooks-style indirection pointer [Flood et al., 2016; Brooks, 1984] to evacuate objects in to-space concurrently. The fourth phase is the concurrent reference updating phase, which also perform a full heap tracing, like the stop-the-world reference updating phase, but runs concurrently, under the help of the Brooks-style indirection pointers.

By performing the marking, evacuation and pointer updating phases run concurrently, Shenandoah GC performs most of the heap scanning work concurrently, 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 [Brooks, 1984]. This requires the Shenandoah collector in JikesRVM to reserve an extra GC word for each Java object [Flood et al., 2016]. 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 updates 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 objects (by following the indirection pointers) while the collector can also evacuate these objects concurrently.

Read Barriers

The read operations of each object field `obj.x`, including the object reference fields and the primitive fields, must 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 extracts 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 points to the new copy of the object. Then by following the indirection pointer, the mutator will always read data from the object's new copy.

Write Barriers

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 update an object while a collector thread is also copying this

```
// class ShenandoahMutator

@Inline
public ObjectReference getForwardingPointer(ObjectReference object) {
  return ForwardingWord.extractForwardingPointer(object);
}

@Inline
public ObjectReference objectReferenceRead(ObjectReference src, Address slot,
    Word metaDataA, Word metaDataB, int mode) {
  return VM.barriers.objectReferenceRead(getForwardingPointer(src), metaDataA,
      metaDataB, mode);
}

// class ForwardingWord

@Inline
public static ObjectReference extractForwardingPointer(ObjectReference oldObject
    ) {
  return oldObject.toAddress().loadWord(FORWARDING_POINTER_OFFSET).and(
      FORWARDING_MASK.not()).toAddress().toObjectReference();
}
```

Figure 3.8: Brooks read barrier

object, the write operation will only perform on the old copy and the new copy may still contain the old data, because the indirection pointer is still pointing to the old copy when the evacuation of this object is still 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 first checks whether 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 `NOT_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 takes the responsibility for forwarding this object. Under such situation, the mutator copies this object and updates the indirection pointer, just like the forwarding process that the collector does.

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

3.6.2 Concurrent evacuation

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

```
@Inline
public ObjectReference getForwardingPointerOnWrite(ObjectReference object) {
  if (brooksWriteBarrierActive) {
    if (isInCollectionSet(object)) {
      ObjectReference newObject;
      Word priorStatusWord = ForwardingWord.attemptToForward(object);
      if (ForwardingWord.stateIsForwardedOrBeingForwarded(priorStatusWord)) {
        // The object is forwarded by other threads
        newObject = ForwardingWord.spinAndGetForwardedObject(object,
            priorStatusWord);
      } else {
        // Forward this object before write
        newObject = ForwardingWord.forwardObjectWithinMutatorContext(object,
            ALLOC_RS);
      }
      return newObject;
    } else {
      return object;
    }
  } else {
    return getForwardingPointer(object);
  }
}

@Inline
public void objectReferenceWrite(ObjectReference src, Address slot,
    ObjectReference tgt, Word metaDataA, Word metaDataB, int mode) {
  ObjectReference newSrc = getForwardingPointerOnWrite(src);
  VM.barriers.objectReferenceWrite(newSrc, tgt, metaDataA, metaDataB, mode);
}
```

Figure 3.9: Brooks write barrier

```
@Override
@Inline
public void processEdge(ObjectReference source, Address slot) {
  ObjectReference oldObject, newObject;
  do {
    oldObject = slot.prepareObjectReference();
    newObject = traceObject(oldObject);
    if (oldObject.toAddress().EQ(newObject.toAddress())) return;
  } while (!slot.attempt(oldObject, newObject));
}
```

Figure 3.10: Concurrent update references

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 always have a valid indirection pointer stored in the header, which will override other status information, e.g. lock and dynamic hash status. So I use a new implementation of object forwarding functions which extends the Java header by one word and stores the indirection pointer in this extra word.

Monitor objects

Monitor objects are handled specially in the JikesRVM implementation of the Shenandoah GC. In JikesRVM, the JVM `monitorenter` and `monitorexit` instructions do not trigger read barriers when locking objects, which can cause inconsistencies in the lock status of the object. I modified the JikesRVM's monitor lock and unlock code to make them trigger the read barriers when necessary.

3.6.3 Concurrent updating references

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

Since the Java program is running during the concurrent reference updating phase, unconditionally updatgin object reference fields may cause races. So instead of unconditionally updating pointers which is used in the previous stop-the-world GCs, I modified the procedure to perform atomic pointer updating, as shown in Figure 3.10, to avoid races.

Object comparisons

Since the mutator can load an address of either the old or new copy of a same object from the heap, simply comparing addresses of objects for the `if_acmpeq` and `if_acmpne` instruction can cause false negatives. I implemented a new object comparison barrier,

```java
@Inline
public boolean objectReferenceCompare(ObjectReference lhs, ObjectReference rhs)
    {
  if (lhs.toAddress().NE(rhs.toAddress()) && getForwardingPointer(lhs).toAddress
      ().NE(getForwardingPointer(rhs).toAddress())) {
    return false;
  } else {
    return true;
  }
}
```

Figure 3.11: Object comparison barrier

as shown in Figure 3.11, to compare object references. Instead 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 loads a' and before loads b', which can still cause false negatives if a and b refer to 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 update the pointers contained by the slot that the CAS is operating on. Under such a situation, the CAS may fail even when there is only one mutator thread updating the slot, since the old value is updated by the collector concurrently.

Figure 3.12 shows the object compare and swap barrier used in Shenandoah GC. Instead of simply performing compare and swap on the old object, after the CAS with the old object failed, another CAS operation is performed where the old value is the indirection pointer of the old object. Now the overall CAS operation still succeeds if the old value is replaced with tha address of a new copy of the same 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, clears their metadata and releases these regions. This phase also runs concurrently without pausing mutators.

3.7 Summary

This chapter identifies, explores and discusses the relationship of the G1 family of garbage collectors. Then each step of the implementation of G1 family of collectors

```java
@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 (!result) {
    result = VM.barriers.objectReferenceTryCompareAndSwap(src,
        getForwardingPointer(old), tgt, metaDataA, metaDataB, mode);
  }
  return result;
}
```

Figure 3.12: Object reference compare and swap barrier

(except C4 GC and ZGC) and the involved GC algorithms are also explained and discussed, as well as a little bit further discussion of the pros and cons for each structural component at an algorithmic level.

However, the pros and cons of several algorithms discussed in this chapter are only discussed abstractly. The real-world GC performance impact caused by these improvements still remains unexplored. So to explore the real-world GC performance, in the next chapter, I perform several performance measurements on a benchmark suite that represents the real-world Java programs, as well as some discussion of the evaluation results.

Chapter 4

Performance Evaluation

Answering the question of GC performance under a real-world setting, this chapter discusses the performance evaluation I undertook for analyzing the Garbage-first family of garbage collectors. This chapter will first describe the software and hardware platform I used for benchmarking. Then the detailed evaluation process and results of pause time, concurrency overhead, and remembered-set footprint will be presented and discussed.

Section 4.1 and 4.2 introduce the experimental platform and software involved for the performance measurements. Section 4.3 discusses the general evaluation method used to evaluate the performance of all the interested metrics. Section 4.4, 4.5 and 4.6 discusses the detailed measurement steps involved to evaluate the GC pause time, concurrency overhead and remembered set size respectively, as well as figures of the results and the resulting discussions.

4.1 The DaCapo Benchmarks

The DaCapo Benchmark Suite is a tool for Java benchmarking and contains a set of open-sourced real-world programs with a high memory load.

The DaCapo benchmarks are frequently used during the development of the G1 family of garbage collectors in Chapter 3, as a validation program to verify the correctness of the collectors under a real-world setting.

I also performed pause time evaluations and concurrency overhead evaluation on all of the following DaCapo Benchmark suites. The benchmarking suites I used for evaluation includes [Blackburn et al., 2006]:

- **antlr** Parses one or more grammar files and generates a parser and lexical analyzer for each.
- **luindex** Uses lucene to indexes a set of documents; the works of Shakespeare and the King James Bible.
- **bloat** Performs a number of optimizations and analysis on Java bytecode files.
- **hsqldb** Executes a JDBCbench-like in-memory benchmark, executing a number of transactions against a model of a banking application.

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.

- **lusearch** Uses lucene to do a text search of keywords over a corpus of data comprising the works of Shakespeare and the King James Bible.
- **pmd** Analyzes a set of Java classes for a range of source code problems.
- **xalan** Transforms XML documents into HTML.
- **eclipse** executes some of the (non-gui) jdt performance tests for the Eclipse IDE.
- **avrora** Simulates a number of programs run on a grid of AVR microcontrollers.
- **sunflow** Renders a set of images using ray tracing.

Except *antlr*, *eclipse*, *fop*, *hsqldb* and *pmd* are coming from DaCapo 2006, all the other benchmarks are coming from the DaCapo version 9.12. Some of the implemented G1 family of collectors can cause some bugs when running on *tomcat*, *batik* and *bloat* benchmarks so I did not include these benchmarks when performing measurements.

By performing evaluations on a wide range of benchmarking suites which represents different class of real-world programs, it is more possible to understand the pros and cons of the G1 family of collectors under a real-world setting.

4.2 Hardware Platform

During the implementation and evaluation of all the G1 family of garbage collectors in Chapter 3, a list of machines with a variety on CPU types, clock, number of processors and the size of cache and memory were involved, as shown in Table 4.1.

As part of the implementation process, by executing the benchmarks on these different machines, I have the ability to statistically verify the correctness of the previously implemented garbage collectors (in Chapter 3) and make sure they perform as intended in a real-world setting.

For further performance evaluation, the "fisher" machine was used for the final benchmarking.

4.3 General Measurement Methodology

I perform both GC pause time and concurrency overhead measurements on the optimized build of the JikesRVM, with 6 different builds for the 6 GCs to be measured respectively. To collect measurement results, I executed all 10 DaCapo benchmarks discussed in Section 4.1 on all 6 collectors, with respect to 4 different heap sizes, 637 MB, 939 MB, 1414 MB, and 1971 MB respectively. This is abnormal since usually heap sizes are different for each benchmark. But by using difference heap sizes for difference benchmarks, it is meaningless to evaluate and compare GC pause time over different heap sizes. So in this thesis I assign each benchmark the same heap size so that I can later compare GC pause time among difference collectors when targeting the same heap size.

Each benchmark suite is executed 10 times for each configuration of $(GC, HeapSize)$ to collect more precise results and avoid the error due to some unexpected environmental fluctuations.

4.3.1 Reducing non-determinisms

The adaptive compiler can have non-deterministic behaviors when performing dynamic compilations and optimizations to the executing Java program. In order to minimize such non-deterministic behaviors and make the program execute faster, I performed the measurement methodology called "warmup replay", which was firstly introduced by Yang et al. [2012], as a replacement of the "pseudoadaptive approach" introduced by Blackburn and Hosking [2004].

This methodology performs an execution of 10 iterations for each benchmark suite to collect run time execution information before the measurement, to assist with more optimized compilation. Then during measurements, after the first iteration of warmups, JikesRVM compiles all methods with the advice information generated previously to avoid any re-compilation behaviors during the following measurement iteration.

I ran all of the 10 benchmark suites discussed in Section 4.1 on each collector for 10 times, with 2 times of warmup execution before the timing iteration. JikesRVM performs warmup-replay compilation at the end of the first warmup iteration.

4.4 Pause Time Evaluation

This section describes the steps took for pause time evaluation as well as all the evaluation results and discussions.

4.4.1 Mutator latency timer

In order to perform more careful analysis on the mutator pause times, instead of simply calculating the time starting from the first stop-the-world phase to the last stop-the-world phase during each GC cycle, I implemented a mutator latency timer to perform a more precise calculation of mutator pause times.

The mutator latency timer contains a static "three-dimensional" array:

```
static long[] LOGS = long[THREAD_ID * EVENT_ID * NUM_LOGS]
```

This array is statically allocated within the VM Space to record the timestamp (in nanoseconds) of each event and each mutator. The first dimension is the thread id of all mutators. The second dimension is the event id. The third dimensional `NUM_LOGS` is the max number of logs of one $(event_id, event_id)$ combination and is currently set to 1024. Particularly under the current context, to measure the pause time for each mutator, two events, `MUTATOR_PAUSE` and `MUTATOR_RESUME` are defined to record the time when a mutator thread starts waiting for GC complete and the time when a mutator gets resumed from a GC pause for execution.

In JikesRVM, every time the program reaches a yieldpoint, the mutator thread checks for GC requests and starts waiting if necessary , which will trigger a `MUTATOR_PAUSE` event if it should be paused. After a stop-the-world cycle is finished, before continuing for further execution, it triggers a `MUTATOR_RESUME` event immediately after gets resumed

At the end of the benchmark execution, the mutator latency timer will dump all the data in the `LOGS` array to the print buffer for further data analysis.

As an output of the analysis of the overhead data, I report the minimum, 50%, and 95 percentile mutator pause time for each GC, each benchmark suite and each heap size.

4.4.2 MMTk harness callbacks

During each iteration of benchmarking, the DaCapo benchmark has several warm-up executions which will run the benchmark suite a few times to warm up the cache and JVM. Then the DaCapo benchmark will start the actual benchmarking run. I use a probe called `MMTKCallback` which will call the `org.mmtk.plan.Plan.harnessBegin` method before the start of the final benchmarking execution and call the `org.mmtk.plan.Plan.harnessEnd` after the final benchmarking execution. Based on this, the two callbacks `harnessBegin` and `harnessEnd` are used to calculate the inform the mutator latency timer to start recording logs or dump all recorded logs.

4.4.3 Results

Table 4.2 shows the results of the GC latency time evaluated on all 6 garbage collectors discussed in Section 3. Specifically, the average, minimum, medium, 95% percentile and maximum GC latency time nanoseconds for each collector and each heap size, as well as the overall GC latency time for each garbage collector were reported.

To present the results more clearly, Figure 4.1 shows the benchmarking results as a set of box plots with log-scaled pause time in nanoseconds as the y-axis. Each subfigure shows the overall GC latency time for each garbage collection algorithm.

	Heap Size = 637 MB				
Pause time (ms)	Mean	Min	Mid	95%	Max
Regional	85.74	37.92	73.26	138.55	416.36
Regional Linear-scan evacuation	110.06	55.28	101.05	160.78	424.84
Regional Concurrent marking	11.36	0.04	1.51	67.06	279.70
Garbage-first Non-generational	9.51	0.04	1.79	46.69	548.71
Garbage-first Generational	10.36	0.04	3.92	44.21	364.37
Shenandoah	4.12	0.04	1.11	9.36	488.65

	Heap Size = 939 MB				
Pause time (ms)	Mean	Min	Mid	95%	Max
Regional	94.40	39.07	82.65	137.98	401.14
Regional Linear-scan evacuation	117.74	53.42	107.33	171.03	545.52
Regional Concurrent marking	18.74	0.04	4.50	74.76	345.44
Garbage-first Non-generational	11.52	0.04	4.46	45.52	803.90
Garbage-first Generational	18.45	0.04	13.63	50.73	1198.33
Shenandoah	3.57	0.04	1.02	10.35	385.64

	Heap Size = 1414 MB				
Pause time (ms)	Mean	Min	Mid	95%	Max
Regional	100.92	38.83	88.16	187.50	449.50
Regional Linear-scan evacuation	131.52	51.83	121.82	197.04	463.21
Regional Concurrent marking	21.18	0.04	4.43	82.69	281.24
Garbage-first Non-generational	13.40	0.03	4.20	43.27	926.35
Garbage-first Generational	27.82	0.04	25.98	54.36	820.84
Shenandoah	5.28	0.04	1.50	13.63	549.97

	Heap Size = 1971 MB				
Pause time (ms)	Mean	Min	Mid	95%	Max
Regional	114.97	39.53	105.31	215.11	415.50
Regional Linear-scan evacuation	150.84	56.66	138.37	234.80	420.24
Regional Concurrent marking	23.98	0.04	4.19	93.37	310.79
Garbage-first Non-generational	15.52	0.04	4.38	46.07	2010.66
Garbage-first Generational	29.20	0.04	25.52	52.58	487.91
Shenandoah	5.55	0.04	1.08	18.06	613.34

Table 4.2: Results of the GC pause time

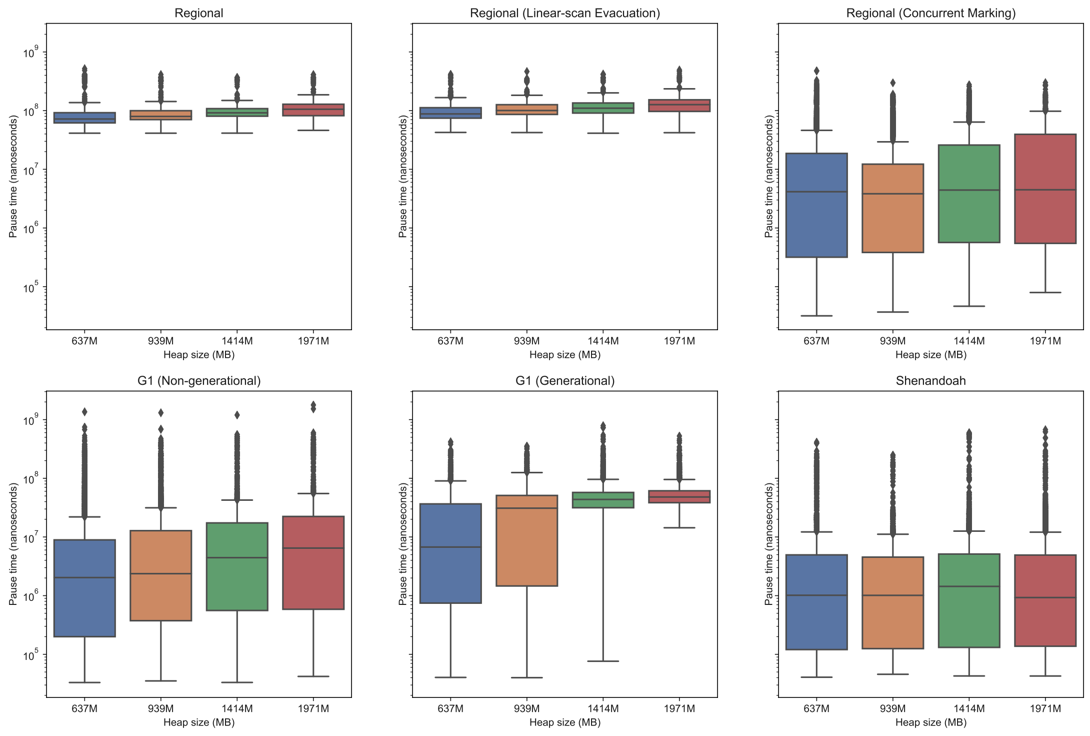

Figure 4.1: Pause times of 6 collectors

Overheads (ns)	Garbage-first Non-generational	Garbage-first Generational
637M	10.26% ±26.26%	5.72% ±15.21%
939M	5.71% ±14.52%	2.24% ± 7.74%
1414M	4.75% ±10.95%	2.38% ± 6.76%
1971M	9.41% ±46.04%	2.21% ± 6.94%
Overall	9.56% ± 7.09%	8.36% ± 6.99%

Table 4.3: Full GCs as a percentage of all GCs

4.4.4 Discussion

As the most simple form of the G1 family of GC, the naive region-based collector, performs stop-the-world GC during each GC cycle. Work for each GC cycle includes marking all live objects and walking over the object graph to evacuate objects that are in a set of selected memory regions. Two full heap tracings are performed during each GC cycle, which make the GC pause time longer. The average pause time for this naive region-based GC generally ranges from 104 to 147 milliseconds and increases as the heap size increases.

By performing linear scan based evacuation, the linear-scan evacuation version of the region-based GC perform a separate linear scan over the collection set to evacuate live objects. In this way, the collector generally has longer pause times, which ranges from 110.1 to 150.8 milliseconds on average and increases as the heap size increases. Linear scan evacuation increases the 95 percentile GC pause time by 15.8%. Although performing linear scan evacuation can increase the GC pause time, this independent phase is an important component for G1 GC and the Shenandoah GC.

After performing concurrent marking in addition to the linear scan based evacuation, the resulting concurrent-marking region-based collector splits the pause for marking into several smaller pauses and performs most of the marking work concurrently without stopping the mutators. This makes the total pause time for a GC smaller and significantly reduces the 95 percentile GC pause time by 58.2%.

The non-generational G1 reduces the collection size to meet a pause time goal of 100 ms. Based on the benchmarking results, at least 95% of the pauses are less than the pre-defined pause time goal. G1 uses remembered sets to update references instead of performing a full heap tracing. For heap sizes of 637 MB, 939 MB, 1414 MB, and 1971 MB, this partial heap scanning technique reduces 95 percentile pause time by 30.4%, 39.1%, 47.7% and 50.7% respectively. This reveals that the remembered sets based references updating has more benefits on larger heaps. However, the full GC becomes more expensive because of the large work required to update all the remembered sets in the heap, which usually results in a pause time ranges from 0.5 s to 1.0 s.

By using the generational G1 GC, young/nursery GCs are usually triggered several times before a major GC happens. Also, nursery GCs are fully stop-the-world and always tries to collect as much nursery regions as possible, as long as the pause time does not exceed the pre-defined pause time goal. This results in the increase of GC pause times. However, the generational collection can largely reduce the probability of G1 GC falling back to full GCs. As shown in Table 4.3, I measured the overall number of full GCs a precentage of all GCs for both non-generational and generational G1 GC on four different heap sizes, with a 95% confidence interval reported as well. Based on the results, with the generational mode, G1 reduces the probability of falling back to full GC by 12.5%.

The Shenandoah GC performs marking, evacuation and reference updating concurrently, this significantly reduces the pause time by 84.1%, compared to the concurrent marking version of the regional collector. Based on the benchmarking results at

Overheads	Average	95% CI
637 M	43.47%	±134.87%
939 M	23.97%	±68.19%
1414 M	11.01%	±23.99%
1971 M	9.33%	±19.93%
Overall	22.00%	±81.90%

Table 4.4: Concurrent marking overhead

least 95% of the GC pauses do not exceed 18 milliseconds. However, full GCs can still result in pauses of around 500 to 600 milliseconds, which are longer than the concurrent-marking regional collector due to the Brooks barrier involved during the evacuation phase, which performance will be discussed in Section 4.5

4.5 Concurrency overhead Evaluation

This section describes the steps took for concurrency overhead evaluation. These concurrency overheads represent the mutator throughput reduction due to the use of related concurrent algorithms (e.g. concurrent marking and evacuation). This section also performs discussions of all the evaluation results.

4.5.1 Methodology

The concurrency percentage overhead is modeled as follows:

$$\text{Overhead} = \frac{|\text{Concurrency overhead} - \text{Mutator time without the specific concurrent phase}|}{\text{Mutator time without the specific concurrent phase}} * 100\%$$

The mutator execution time is calculated as the execution time of the benchmarking program with stop-the-world GC time excluded.

As discussed in Chapter 3, I implemented the Garbage-first family of collectors by performing progressive improvements over a simple region-based collector. In this way, after performing an algorithmic improvement over a collector, we can measure the overhead of the newly involved concurrent jobs or other technologies by comparing the benchmarking results of the old collector and the new collector.

As an output of the analysis of the overhead data, the average Concurrency overhead for each GC, each benchmark suite, each heap size and each concurrent phase this project used are reported as well as their corresponding 95% confidence interval. The overall average overhead and its 95% confidence interval for each concurrent job are also reported.

Overheads	Average	95% CI
637 M	66.84%	±146.32%
939 M	53.76%	±99.08%
1414 M	62.19%	±117.91%
1971 M	55.82%	±133.88%
Overall	59.65%	±125.98%

Table 4.5: Concurrent remembered-set refinement overhead

4.5.2 Concurrent marking overhead

Table 4.4 shows the overheads of concurrent marking as well as their 95% confidence interval on different heap sizes. Based on the evaluation data, the concurrent-marking has an overhead of 22.0% on average.

The SATB barrier used by the concurrent marking algorithm is a deletion barrier. Furing concurrent marking, the barrier will trace and mark all deleted nodes (i.e. the old object field when performing assignment `obj.x = y`) in the object graph. A major difference between the SATB barrier used in these region-based collectors and the concurrent mark-sweep GC is that the concurrent mark-sweep GC only marks objects when tracing an object (the mark data is usually stored in the object header). But the G1 family of collectors have to count the live bytes for each region to assist with further collection set selection. Also, these G1 family of collectors use off-heap mark table instead of object header to store liveness data. For these reasons, a lot of atomic operations are involved during concurrent marking which further reduces the mutator throughput, compared to the concurrent mark-sweep GC.

4.5.3 Concurrent remembered-set refinement overhead

Table 4.5 shows the overheads of the Concurrent remembered-set refinement overhead as well as their 95% confidence interval on different heap sizes.

Node that this result represents the concurrency overhead of both remembered-set barriers and concurrent remset refinements. The design of the remembered-set barriers and the remset refinement process follows the original design of G1 that only 1 thread is used to process the dirty card buffer and it only awakes for processing when the card buffer is full.

Based on the measurement results, the overhead of remembered-set refinement is not low, which is 59.7% on average. This is because the number of threads used for remset refinement is not enough and the dirty card buffer always becomes full. Under such situation, mutators have to take part of the responsibility to process cards in their local buffer, which significantly reduces its throughput.

A possible fix, which has already been introduced into the OpenJDK's G1 implementation, is to spawn more threads for remset refinement, and refinement threads can start processing cards earlier, not necessarily need to wait until the global card

Overheads	Average	95% CI
637 M	99.71%	±118.92%
939 M	92.63%	±98.99%
1414 M	82.15%	±87.11%
1971 M	66.98%	±80.07%
Overall	85.46%	±100.44%

Table 4.6: Concurrent evacuation overhead

buffer is full.

4.5.4 Concurrent evacuation overhead

Table 4.6 shows the overheads of the Concurrent evacuation overhead as well as their 95% confidence interval on different heap sizes. On average, by using concurrent evacuation, the concurrency overhead of Shenandoah GC is increased by 85.5%.

This is a significantly high overhead. The reason for causing this is the use of "use barriers" which insert a barrier every time the JVM wants to access and use an object reference. In addition, as described in Section 3.6, during the concurrent evacuation phase, an extra barrier is used for every object comparison operation to ensure the correct comparison between the forwarded and unforwarded pointer of the same object reference, which further increases the concurrency overhead.

Another reason for causing this high overhead is that I could not fix a data race problem happens during concurrent evacuation, due to the time scope of this project. Instead I implemented a work-around to prevent this problem from happening. This work-around has bad mutator performance since it requires addition works (e.g. forward objects if necessary) to be done in read barriers. Although the overhead of this work-around is not measured, it is expected to contribute to most of the concurrent evacuation overhead shown in Table 4.6. Plans to fix this problem will be discussed in Section 5.1 as a part of the future work.

However, in order to perform concurrent evacuation and reference updating, the handling of forwarded and unforwarded pointers is necessary. One possible improvement is to use "colored pointers" to ensure the CPU will always access the new version of the object pointer before using this pointer, instead of go through the indirection pointer every time the mutator access the object.

4.6 Remembered-Set Size

As part of the evaluation of the G1 collector, I measured the remembered-set footprint of G1 GC.

The implementation of remembered-set follows the design of the original Open-JDK implementation, which uses a list of `PerRegionTable` as remembered-set for each region. Each `PerRegionTable` remembers cards in one foreign region. `PerRegionTable` is implemented as a bit table where each bit corresponds to a card in the corresponding foreign region.

Under such implementation, theoretically the space complexity for remembered-sets is $O(N^2)$ where N is the number of regions in the heap. Because each of total N regions has its own remembered-set and each remembered-set consists of $N-1$ `PerRegionTables` to remember cards in other $N-1$ regions. However, the practical space performance of such remembered-set structure has never been formally measured.

Due to the same remembered-set structure, measurements for the remembered-set footprint of JikesRVM's G1 implementation can reflect the footprint of the OpenJDK's implementation, which can help us understand the space performance of G1's remembered-sets under a real-world setting.

4.6.1 Evaluation metrics

In order to measure the remembered-set footprint carefully, two metrics are proposed to reflect the space performance of remembered-sets:

Committed Memory Ratio is the ratio of the memory allocated for building remembered-sets versus the total committed memory at some specific execution point. This metric reflects the proportion of the heap that remembered-sets are actually take up.

$$\text{Committed Memory Ratio} = \frac{\text{Committed memory for remset}}{\text{Total committed memory}} * 100\%$$

Utilization Ratio is the ratio of the memory (in bits) actually used for remembered-sets to remember the cards, versus the total memory allocated for remembered-sets. This metric reflects the proportion of the remembered-sets that is actually in use and not be wasted.

$$\text{Utilization Ratio} = \frac{\text{Bits actually used for store cards}}{\text{Committed memory for remset in Bits}} * 100\%$$

4.6.2 Results & discussion

Remembered-set size is always changing during the execution of the program. Plus, as the remembered-set refinement thread may still processing cards, the remembered-set can be incomplete and may not remember all the corresponding cards. Because in such situation some cards are still waiting to be processed and write to some remembered-sets.

For this reason, I measure the remembered-set footprint at the start of each GC pause, immediately after the stop-the-world remembered-set refinement is finished. At this time the remembered-set footprint reaches a steady state, the remembered-set is complete and contains all the corresponding cards under current heap state.

Heap Size = 637 MB

Footprint	Committed Memory			Utilization		
	Min	Mean	Max	Min	Mean	Max
Garbage-first Non-generational	2.88%	8.30% ±6.09%	30.64%	0.08%	2.35% ±7.60%	27.97%
Garbage-first Generational	2.77%	8.69% ±6.68%	23.44%	0.09%	2.14% ±4.69%	19.70%

Heap Size = 939 MB

Footprint	Committed Memory			Utilization		
	Min	Mean	Max	Min	Mean	Max
Garbage-first Non-generational	2.61%	8.98% ±9.08%	38.23%	0.07%	1.82% ±6.09%	21.53%
Garbage-first Generational	2.26%	9.77% ±7.63%	26.08%	0.08%	2.19% ±4.81%	16.98%

Heap Size = 1414 MB

Footprint	Committed Memory			Utilization		
	Min	Mean	Max	Min	Mean	Max
Garbage-first Non-generational	2.36%	7.98% ±6.10%	36.76%	0.03%	1.11% ±3.67%	21.50%
Garbage-first Generational	3.18%	9.85% ±6.93%	23.62%	0.08%	2.00% ±4.02%	22.55%

Heap Size = 1971 MB

Footprint	Committed Memory			Utilization		
	Min	Mean	Max	Min	Mean	Max
Garbage-first Non-generational	2.25%	7.98% ±6.02%	37.00%	0.03%	0.91% ±2.76%	20.85%
Garbage-first Generational	3.22%	10.15% ±6.76%	21.72%	0.08%	2.81% ±6.64%	22.00%

Overall

Footprint	Committed Memory			Utilization		
	Min	Mean	Max	Min	Mean	Max
Garbage-first Non-generational	2.25%	8.36% ±6.99%	38.23%	0.03%	1.74% ±6.09%	27.97%
Garbage-first Generational	2.26%	9.56% ±7.09%	26.08%	0.08%	2.27% ±5.14%	22.55%

Table 4.7: Remembered set footprint

As shown in Table 4.7, I measured both proposed metrics on both non-generational and generational G1 GC, with four different heap sizes. For each kind of G1 GC and each heap size, I report the minimum value, maximum value and mean value with 95% confidence interval for both committed memory ratio and utilization ratio.

Based on the footprint data, we can see that for generational G1 Gc, an average 9.6% of the committed memory is allocated for building remembered-sets, with a maximum proportion of 38.2%. For non-generational G1 GC the proportion of committed memory is 8.4% on average. Which means that remembered-sets are taking up too much memory in the heap. Also, the memory usage (i.e. utilization) for remembered-sets is pretty low, only 2.3% of the memory allocated for remembered-sets is actually used for remembering cards. Taking the high committed memory ratio into consideration, this means that the space efficiency of the `PerRegionTable` based remembered-sets is extremely low. Hence a lot of optimization works should be done to further reduce the memory waste of remembered-sets.

4.7 Summary

This chapter discusses the measurement methodology and evaluation results of the GC performance of the G1 family of garbage collectors. This includes the evaluation of GC pause times and overhead of several concurrent phases. Also, the phenomenon revealed in the measurement results is carefully discussed.

On the one hand, based on the measurement results, we can see that linear scan based evacuation increases the work for each GC. Using concurrent marking, concurrent evacuation or remembered set based partial heap scanning can significantly reduce the pause time for each GC. On the other hand, using concurrent algorithms can largely increase the mutator throughput reduction.

Also, based on the measurement results, the generational G1 and Shenandoah GC shows some disappointing performances on GC pause time and concurrency overheads respectively. However, the time scope of this project is limited. In this way, there are still much optimization jobs and other works to do in the future, which will be discussed in Chapter 5.

Chapter 5

Conclusion

This thesis aimed to identify and explore the underlying relationship among the G1 family of garbage collectors, as well as analyze the performance impact due to each structural component. The explorations and discussions in the previous chapters have successfully demonstrated that the relationship among the G1 family of collectors exists and can impact the GC performance in both positive and negative ways. Hence the pros and cons of each algorithm and the cause of these phenomena are discussed based on the evaluation results.

5.1 Future Work

Although this thesis has come to an end, the research on this topic is far from finished. There is still much work to do after this research project. In general, a few typical future works are: 1. Resolve a data race problem for Shenandoah GC. 2. Perform some optimizations, 3. Exploring C4 GC and ZGC. 4. Further G1 related GC research.

5.1.1 Race problem for Shenandoah GC

Currently, the implementation of Shenandoah GC has a data race problem. The mutator is not locking and releasing monitors properly during the concurrent evacuation phase of the Shenandoah GC, when there can be two different copies of an object exist in the heap.

I cannot solve this issue due to the time scope of this thesis. But currently, a workaround is implemented in the Shenandoah GC, with a lower mutator performance. This explains the bad performance of the concurrent evacuation overhead we evaluated in Chapter 4. As the most urgent problem I am facing, I plan to solve this issue immediately after this project.

5.1.2 Optimizations

As part of the mutator latency results discussed in Chapter 4, the generational G1 currently does not reveal too much performance gain compared to the non-generational G1, except the decrease of full GC ratio. It is expected to have more optimizations and parameter tuning to the generational G1 to make further performance improvements.

I am also considering the possibility to make the implemented Garbage-first and Shenandoah GC become production ready. One major missing part is the optimization since the development of these garbage collectors was following the original design of these collectors and did not have too many optimizations. After performing some optimizations on these collectors as well as some additional correctness verification, the collector can have the possibility to become production ready.

5.1.3 C4 GC and ZGC

Due to the time scope of this project, I did not implement and measure C4 GC and ZGC. In addition, since JikesRVM and MMTk only support the 32bit address space but the pointer coloring process in ZGC requires the 64bit address space, which makes the implementation of ZGC more difficult.

However, as the latest member of the Garbage-first family of collectors, C4 and ZGC have better pause time performance than G1 GC and Shenandoah GC generally. By performing most of the GC work concurrently, the pause time of C4 and ZGC are not proportional to the heap size and are expected to be less than 10 milliseconds even targeting 100 GB heaps according to Liden and Karlsson [2018]. In this way, these two collectors are extremely worth for an exploration.

A detailed plan for resolving several hardware incompatibilities should be done in the future, before starting the implementation of these two collectors.

5.1.4 Future G1 related GC research

The Garbage-first family of garbage collectors have been proved to have high GC performance, in terms of GC latency and concurrency overheads. However, the measurement results are still not perfect, which means there is a lot more can be done to make further GC performance improvements.

For instance, a generational extension can be applied to the Shenandoah GC to collect young garbage as early as possible to decrease the frequency of falling to full GCs. This generational mode involves a remembered set to remember object pointers in mature space pointing to the nursery space. In addition to using the table-based remembered sets used in G1, the buffer-based remembered sets introduced by Blackburn and McKinley [2008] can also be explored as a comparison to G1's remembered sets.

To summarize, there are still many details of G1 related garbage collection algorithms can be explored. By performing more improvements over the existing Garbage-first family of collectors (e.g. the Shenandoah GC), it is highly possible for GC performance to have more improvements.

5.2 Summary

This thesis is aimed to identify and explore the underlying relationship among the G1 family of garbage collectors, implement them as a series of collectors to reflect such relationships and analyze GC performance impact of different algorithmic components.

As discussed in chapters 3, the potential relationships among the G1 family of collectors are successfully identified and discussed. Based on these relationships, this thesis produces the first implementation of the G1 family of collectors that reflect the underlying algorithmic relationships. Most of my implementations result in a reasonable or even exceptional performance in terms of GC pause time and concurrency overheads. Based on these implementations and measurement results, the pros and cons, as well as their underlying reasons, are explained and discussed. Hence, the explorations performed in this thesis can inspire GC designers to reconsider the design and structure of region-based GC algorithms to make further valuable algorithmic improvements.

In conclusion, the relationships among the G1 family of collectors exist and have an impact on GC performance in many ways, including negative performance impacts. This means that there is still a lot to improve and more research should be done in this area.

Bibliography

Alpern, B.; Augart, S.; Blackburn, S. M.; Butrico, M.; Cocchi, A.; Cheng, P.; Dolby, J.; Fink, S.; Grove, D.; Hind, M.; et al., 2005. The jikes research virtual machine project: building an open-source research community. *IBM Systems Journal*, 44, 2 (2005), 399–417. (cited on page 5)

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 pages 3 and 6)

Blackburn, S. M.; Garner, R.; Hoffmann, C.; Khang, A. M.; McKinley, K. S.; Bentzur, R.; Diwan, A.; Feinberg, D.; Frampton, D.; Guyer, S. Z.; Hirzel, M.; Hosking, A.; Jump, M.; Lee, H.; Moss, J. E. B.; Phansalkar, A.; Stefanović, D.; VanDrunen, T.; von Dincklage, D.; and Wiedermann, B., 2006. The dacapo benchmarks: Java benchmarking development and analysis. *SIGPLAN Not.*, 41, 10 (Oct. 2006), 169–190. doi:10.1145/1167515.1167488. http://doi.acm.org/10.1145/1167515.1167488. (cited on page 35)

Blackburn, S. M. and Hosking, A. L., 2004. Barriers: Friend or foe? In *Proceedings of the 4th international symposium on Memory management*, 143–151. ACM. (cited on page 37)

Blackburn, S. M. and McKinley, K. S., 2008. Immix: a mark-region garbage collector with space efficiency, fast collection, and mutator performance. In *ACM SIGPLAN Notices*, vol. 43, 22–32. ACM. (cited on pages 8 and 50)

Brooks, R. A., 1984. Trading data space for reduced time and code space in real-time garbage collection on stock hardware. In *Proceedings of the 1984 ACM Symposium on LISP and functional programming*, 256–262. ACM. (cited on pages 14 and 28)

Collins, G. E., 1960. A method for overlapping and erasure of lists. *Communications of the ACM*, 3, 12 (1960), 655–657. (cited on page 6)

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 1, 8, 9, 10, 14, 21, and 26)

Fenichel, R. R. and Yochelson, J. C., 1969. A lisp garbage-collector for virtual-memory computer systems. *Communications of the ACM*, 12, 11 (1969), 611–612. (cited on page 8)

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 pages 1, 10, 14, 27, and 28)

GAY, D. AND AIKEN, A., 1998. *Memory management with explicit regions*, vol. 33. ACM. (cited on page 10)

HOSKING, A. L. AND HUDSON, R. L., 1993. Remembered sets can also play cards. *OOPSLAGC OOP93]. Available for anonymous FTP from cs. utexas. edu in/pub/garbage/GC93*, (1993). (cited on page 23)

LIDEN, P. AND KARLSSON, S., 2018. Zgc: A scalable low-latency garbage collector. http://openjdk.java.net/jeps/333. (cited on pages 1, 10, 14, and 50)

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

MCCARTHY, J., 1960. Recursive functions of symbolic expressions and their computation by machine, part i. *Communications of the ACM*, 3, 4 (1960), 184–195. (cited on page 7)

NGUYEN, K.; FANG, L.; XU, G.; AND DEMSKY, B., 2015. Speculative region-based memory management for big data systems. In *Proceedings of the 8th Workshop on Programming Languages and Operating Systems*, 27–32. ACM. (cited on page 2)

TENE, G.; IYENGAR, B.; AND WOLF, M., 2011. C4: The continuously concurrent compacting collector. *ACM SIGPLAN Notices*, 46, 11 (2011), 79–88. (cited on pages 1, 9, and 14)

YANG, X.; BLACKBURN, S. M.; FRAMPTON, D.; AND HOSKING, A. L., 2012. Barriers reconsidered, friendlier still! In *ACM SIGPLAN Notices*, vol. 47, 37–48. ACM. (cited on pages 11 and 37)

YUASA, T., 1990. Real-time garbage collection on general-purpose machines. *Journal of Systems and Software*, 11, 3 (1990), 181–198. (cited on pages 14, 15, and 20)