JOHANNES KEPLER
UNIVERSITY LINZ

Author
**Christian Aistleitner**
**B.Sc.**

Submission
**Institute for**
**System Software**

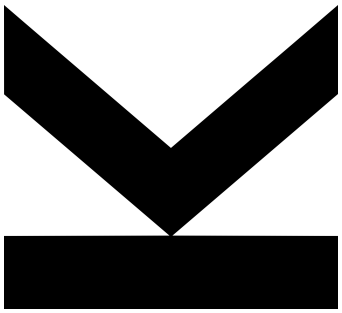
Supervisor
**Prof. Dr. Dr. h.c.**
**Hanspeter Mössenböck**

Co-Supervisors
from Oracle Labs
**Dr. Peter Hofer**
**Dr. Christian Häubl**
**Dr. Christian Wirth**

January 2024

# A new Mark-and-Compact GC for GraalVM Native Image

Master Thesis with Oracle Labs



Master Thesis
to obtain the academic degree of
Diplom-Ingenieur
in the Master's Program
Computer Science

**JOHANNES KEPLER**
**UNIVERSITY LINZ**
Altenbergerstraße 69
4040 Linz, Austria
www.jku.at
DVR 0093696

# Abstract

Containerized cloud deployments with tight resource limits are becoming increasingly popular. To address this trend, GraalVM Native Image provides ahead-of-time compilation of Java applications into standalone executable binaries. These are ideal for cloud deployments as they offer fast startup speed, low memory footprint, and small packaging size.

The garbage collector of GraalVM Native Image is written in Java too and, like the rest of the Java runtime system, it is part of the generated binary. It uses a simple generational Stop-and-Copy algorithm for collecting unreachable objects. New objects are allocated in the young generation and promoted to the old generation after surviving multiple minor collection cycles or a single major collection cycle. In the course of a collection cycle, every reachable object is copied, which increases the memory footprint during scavenging.

This thesis describes the implementation of a new compacting garbage collector with an improved memory footprint. It makes use of the Mark-and-Compact algorithm for collecting unreachable objects in the old generation. Furthermore, this thesis presents a performance evaluation of the new approach using existing benchmarks.

# Kurzfassung

Immer mehr Anwendungen laufen mit engen Ressourcenbeschränkungen in der Cloud. Um diesem Trend gerecht zu werden, bietet GraalVM Native Image eine Kompilierung von Java-Anwendungen im Voraus zu eigenständig ausführbaren Binärdateien. Diese eignen sich dank ihres schnellen Starts, ihres geringen Speicherbedarfs und einer kleinen Paketgröße ideal für Cloud-Bereitstellungen.

Der Garbage Collector von GraalVM Native Image ist ebenfalls in Java geschrieben und wie der Rest des Java-Laufzeitsystems Teil der generierten Binärdatei. Zur Speicherbereinigung verwendet dieser Garbage Collector einen einfachen generationellen Stop-and-Copy-Algorithmus. Neue Objekte werden in der Young-Generation angelegt und an die Old-Generation weitergegeben, nachdem sie mehrere Bereinigungzyklen überstanden haben. Im Verlauf eines Bereinigungzyklus wird vorübergehend von jedem erreichbaren Objekt eine Kopie erstellt, was den Speicherbedarf erhöht.

Diese Arbeit beschreibt die Umsetzung eines neuen Garbage Collectors, welcher den Speicherbedarf weiter reduziert. Der neue Garbage Collector nutzt den Mark-and-Compact-Algorithmus, um den Speicher in der Old-Generation aufzuräumen. Darüber hinaus präsentiert diese Arbeit eine Performance-Auswertung des neuen Ansatzes anhand vorhandener Benchmarks.

# Contents


**1 Introduction** **5**

1.1 Problem and Motivation . . . . . 5

1.2 Proposed Solution . . . . . 6

1.3 Contribution . . . . . 6

1.4 Structure of this Thesis . . . . . 7

**2 Background** **8**

2.1 Garbage Collection . . . . . 8

2.1.1 Stop-and-Copy algorithm . . . . . 9

2.1.2 Mark-and-Compact algorithm . . . . . 11

2.2 GraalVM . . . . . 13

2.2.1 GraalVM Native Image . . . . . 14

**3 Fundamentals of the Substrate VM** **17**

3.1 Native Memory Access from Java . . . . . 17

3.2 Image Heap versus Java Heap . . . . . 18

3.3 Chunks . . . . . 19

**4 Our New Mark-and-Compact GC** **20**

4.1 Architecture . . . . . 20

4.2 Generational Heap . . . . . 23

4.3 Minor Collection . . . . . 24

4.4 Major Collection . . . . . 26

4.4.1 Mark Phase . . . . . 26

4.4.2 Plan Phase . . . . . 29

4.4.3 Reference Fixup Phase . . . . . 35

4.4.4 Compaction Phase . . . . . 36

**5 Performance Evaluation** **37**
5.1 Benchmarks . . . . . 37
5.2 Results . . . . . 40

**6 Related Work** **42**
6.1 .NET Runtime . . . . . 42
6.2 HotSpot JVM . . . . . 42

**7 Conclusion** **43**
7.1 Summary . . . . . 43
7.2 Future Work . . . . . 44

# 1 Introduction

This chapter provides an overview of the problem at hand and introduces the proposed solution that is being implemented and evaluated within the scope of this master's thesis. Additionally, it offers a preview of what to expect in the upcoming chapters for a clearer understanding of the thesis structure.

## 1.1 Problem and Motivation

GraalVM Native Image [1] provides ahead-of-time (AOT) compilation of Java code to native binaries that run without a pre-installed Java Runtime Environment (JRE). The garbage collector (GC) is written in Java too and, like the rest of the Java runtime system, it is part of the AOT compiled binary.

The current default GC, the Serial GC, uses a generational Stop-and-Copy [2] algorithm. Newly created objects and short-lived objects reside in the young generation, while long-lived objects reside in the old generation. Objects are being promoted from the young to the old generation if they survive multiple minor collections or a single major collection. In the course of a collection cycle, every reachable object is copied to a new location, and a forwarding pointer is written to the original location.

This copying approach requires up to twice the amount of heap memory during major collections. As a consequence, the GC needs to reserve additional memory that is not available to the actual application, causing an increased memory footprint. Furthermore, `OutOfMemoryError` exceptions are predominantly thrown during collection cycles rather than at the point of large object allocations.

## 1.2 Proposed Solution

The objective of this thesis is to implement a new GC that provides an improved memory footprint while still maintaining high throughput. The basis of the new GC should follow established best practices. This includes a Mark-and-Compact [2] algorithm for the old generation and a Stop-and-Copy algorithm with aging for the young generation.

The work represents an evolutionary step that can build on the existing source code of the Serial GC by adapting its components.

Tailored to the existing memory management concepts of GraalVM Native Image, the new approach should take advantage of the fact that the heap is not a contiguous memory region but is separated into chunks. For example, the GC can choose to compact only those chunks that contain a significant number of dead objects.

The target applications of this new GC are microservices and function-as-a-service applications that are mostly single-threaded and need to run with tight memory size constraints, e.g. in a containerized environment.

## 1.3 Contribution

A goal of this thesis is to make an open-source contribution to the official public GraalVM repository [3] hosted on GitHub. The pull request covers the implementation of the new garbage collection approach while replacing the existing one.

## 1.4 Structure of this Thesis

After this introduction, Chapter 2 provides background information about the technologies this work is based on. Chapter 3 describes fundamental concepts that are specific to the project environment. The implementation of the project is being presented in Chapter 4. This includes technical descriptions of implementation details. Chapter 5 features the performance evaluation of our new GC using existing benchmarks. Lastly, we will have a look at related work in Chapter 6 before finishing this thesis with a conclusion in Chapter 7.

# 2 Background

This chapter introduces fundamental concepts that are relevant to this thesis. It provides the necessary background information and a broader view of the topic of memory management in general. The chapter begins with an introduction to the core principles of garbage collection and concludes with a description of GraalVM.

## 2.1 Garbage Collection

Garbage collection is a critical aspect of modern programming languages like Java. A GC automatically identifies and reclaims memory that is no longer in use and thus relieves developers from the burden of manually deallocating memory.

One of the early influential figures in the development of the garbage collection concept was John McCarthy, who is known for his work on the LISP programming language [4] in the late 1950s. LISP includes automatic memory management features that can be considered as the first implementation of a GC.

Since then, many researchers and developers have contributed to the topic of garbage collection algorithms and implementations [2][5]. Shifting our focus to the present day, we see that all modern programming languages are using garbage-collected memory, and there are GC implementations with almost neglectable performance overhead.

Significant benefits of automated memory management systems that powered the success story of the GC concept include:

- Prevention of memory leaks while ensuring efficient memory usage
- Simplification of the development process and improved program stability

### 2.1.1 Stop-and-Copy algorithm

The Stop-and-Copy algorithm is one of the oldest and simplest approaches used to reclaim memory occupied by unreachable objects [6][7]. It divides the memory into two semi-spaces of equal size:

- The *from-space* holds all objects during program execution. New objects are also allocated here.
- The *to-space* to which the live objects will be copied during the collection process. It is not in use during the execution of the actual program.

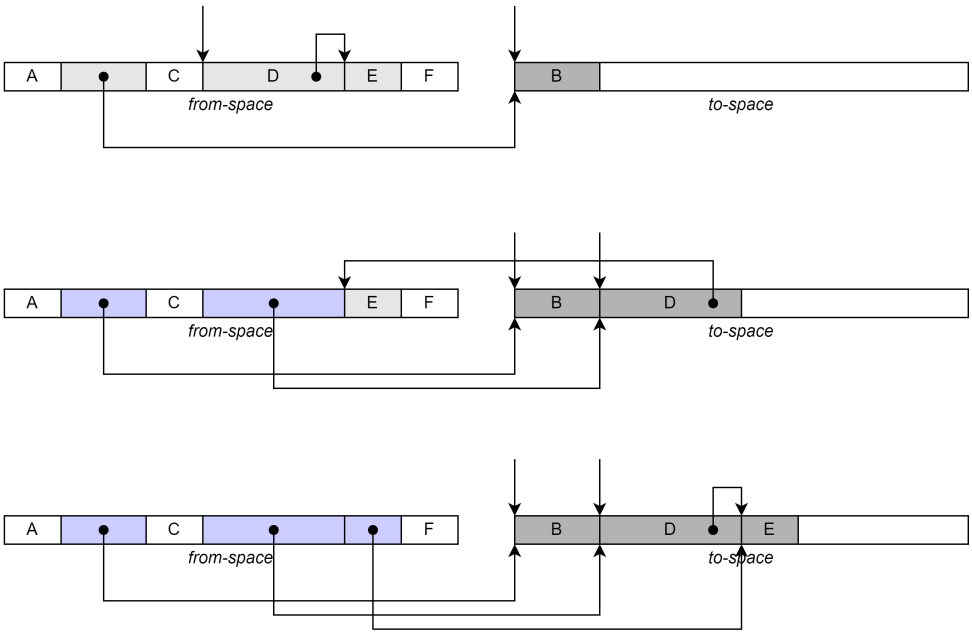
As depicted in Figure 2.1, the algorithm follows these basic steps:

1. Halt the execution of all program threads.
2. Traverse the object graph starting from root pointers.
   For each reference edge do the following:
   a) If the referenced object has not been copied yet,
   copy it to the to-space, mark it as stale, and install a forwarding pointer in the stale object that points to the new location.
   b) If the referenced object has already been copied,
   read its forwarding pointer to retrieve its new location.
   c) Update the outgoing reference in the holder object.
3. Reset the from-space and swap the two semi-spaces so that the former to-space becomes the new from-space.
4. Resume the execution of the program.

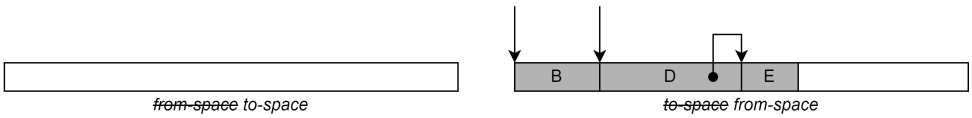
Advantages of the Stop-and-Copy algorithm include its simplicity, efficiency in terms of reducing fragmentation, and high throughput as it takes only a single pass through the heap. However, it does have the major drawback of requiring additional memory to maintain the two semi-spaces. This results in an increased memory footprint as only half of the memory is available for the actual application.

**Step 1:** Halt the execution of all program threads.


**Step 2:** Traverse the object graph starting from root pointers and copy live objects.




**Step 3:** Reset the from-space and swap labels of the two semi-spaces.




**Step 4:** Resume the execution of the program.

**Figure 2.1:** Stop-and-Copy algorithm

### 2.1.2 Mark-and-Compact algorithm

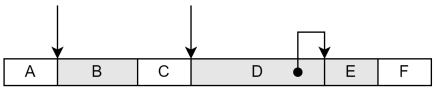
The Mark-and-Compact algorithm identifies live objects before reclaiming memory by compacting them in within a single memory space. As shown in Figure 2.2, the algorithm follows these steps:

1. Halt the execution of all program threads.
2. Traverse the object graph starting from the roots. For each object node:
    a) Mark the object as live by setting a flag in the object header.
3. Iterate through the heap space. For each marked object:
    a) Compute the new location of the object after compaction.
    b) Save the computed address as the relocation pointer in the object header or in a separate data structure.
4. Iterate through the roots and the heap space. For each reference:
    a) Retrieve the referenced object's relocation pointer.
    b) Update the outgoing reference in the holder object.
5. Iterate through the heap space. For each object with a relocation pointer:
    a) Move the object to its new location.
6. Resume the execution of the program.

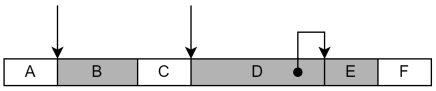
While this algorithm has a much smaller memory footprint as it does not have to reserve additional memory for copying objects, it has the drawback of requiring a mark phase and three additional passes through the heap compared to the Stop-and-Copy algorithm. This makes it noticeably slower.

Furthermore, the object header must include an additional flag to mark the object as live and a field to store the relocation pointer, if it is not stored in a separate data structure. This increases the object header size which should be as small as possible for efficiency reasons.

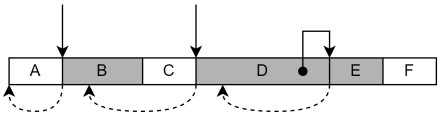
**Step 1:** Halt the execution of all program threads.



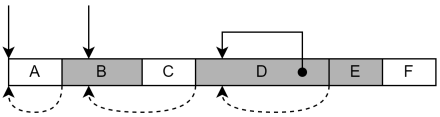

**Step 2:** Mark reachable objects.



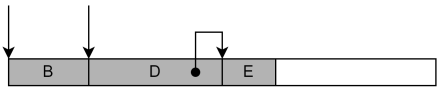

**Step 3:** Define the relocation pointers.




**Step 4:** Adjust object references.




**Step 5:** Move objects to their new location.




**Step 6:** Resume the execution of the program.

**Figure 2.2:** Mark-and-Compact algorithm

## 2.2 GraalVM

GraalVM [8] is an advanced, high-performance, multi-language JDK developed by Oracle Labs. It is based on the HotSpot JVM and uses the Graal compiler [9] as its dynamic just-in-time (JIT) compiler to translate Java bytecode into highly-optimized machine code.

The Graal compiler is written in Java and interacts with the JVM using the low-level JVM Compiler Interface (JVMCI). It is designed to improve the performance of Java applications by using multiple advanced optimization algorithms and techniques, like aggressive and polymorphic inlining or partial escape analysis.

Furthermore, GraalVM's language implementation framework named *Truffle* [10] enables support for a variety of guest languages beyond Java, as depicted in Figure 2.3. This also includes interoperability between these languages, allowing for polyglot applications.

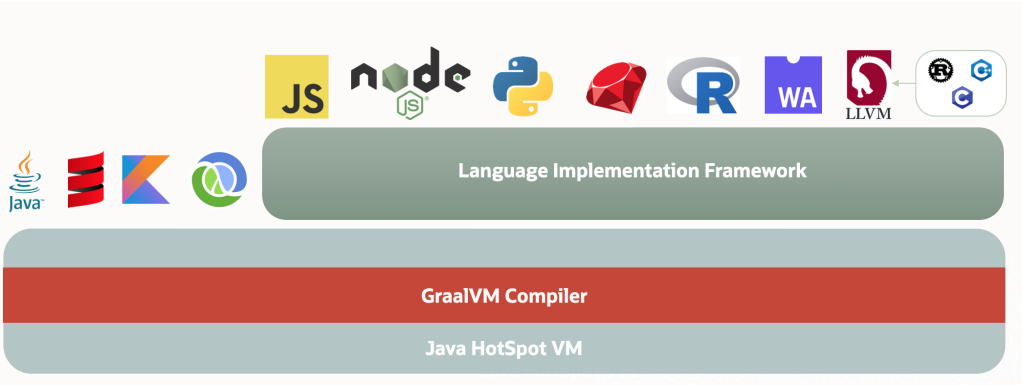



**Figure 2.3:** GraalVM architecture [11]

GraalVM is available in two distributions:

- *GraalVM Community Edition* (CE)
  which is open-source software and the primary focus of this thesis.
- *GraalVM Enterprise Edition* (EE)
  which provides advanced features and optimizations but requires a paid subscription.

### 2.2.1 GraalVM Native Image

GraalVM also provides an ahead-of-time (AOT) compiler for translating Java applications at build time into stand-alone, self-contained executable binaries [12]. This technology is called *GraalVM Native Image* and is also being developed by Oracle Labs.

The generated binaries include application classes, library classes, and the so-called *Substrate VM* [13] which covers all necessary runtime system components like the garbage collector. This allows the application to run without a pre-installed Java Runtime Environment (JRE).

Figure 2.4 shows a high-level overview of the steps involved in building a native image [14]. First, GraalVM Native Image finds all reachable classes, methods, and fields using static analysis. Only used code will be included in the generated executable. Then the application is initialized by executing all initializer blocks. We can do that as most class initialization code does not depend on any external input such as configuration files. Finally, a snapshot of the application state is made and all source code is compiled to machine code. Objects created by initializer blocks at build time are available at run time in the so-called image heap. More on that later in Section 3.2.

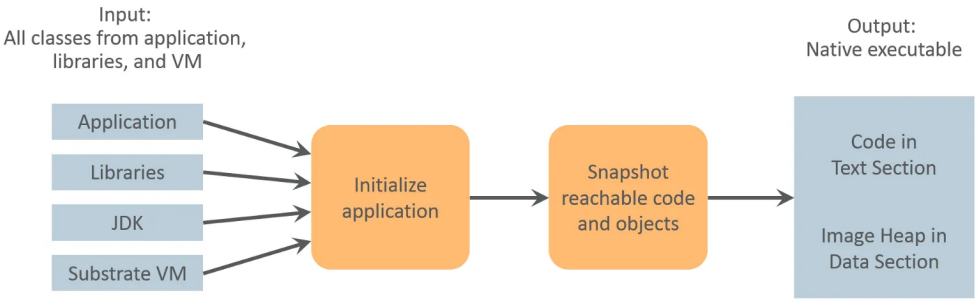



**Figure 2.4:** Native Image [14]

GraalVM Native Image provides a new way to run Java applications and also promotes modern cloud-native deployment [1]. Figure 2.5 shows the main tradeoffs between JIT and AOT compilation [15]. Key advantages of AOT compilation over traditional means include a faster startup speed, reduced memory footprint during execution, and small package size. However, this comes at the cost of reduced peak throughput and increased latency.

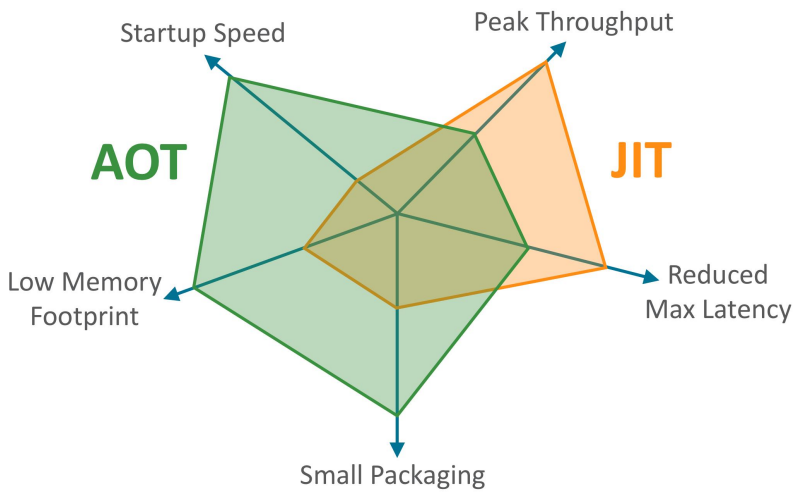



**Figure 2.5:** Key advantages of AOT compilation [15]

GraalVM Native Image is well suited for microservices that are deployed in a containerized environment [16]. Modern software deployments focus on scalability, since demand for a given service can be quite volatile. Therefore, it is important to be able to quickly scale horizontally on a high load, i.e., by adding more instances when demand is high. The fast startup speed and the small packaging of AOT compilation is exactly what we need in such a case. Additionally, the improved memory footprint reduces costs.

AOT compilation is especially interesting in the context of Function-as-a-Service (FaaS) platforms, since a fast startup time is an even more critical factor there in order to handle invocations as fast as possible.

Furthermore, it is well integrated into the GraalVM ecosystem and even supports polyglot applications to be compiled into stand-alone executables.

There are three garbage collector implementations available to choose from in the Substrate VM [17], and each has its advantages in certain use cases:

- **Serial GC** (default)
  Single-threaded and optimized for low memory footprint and small Java heap sizes. It supports advanced GraalVM features such as isolates (multiple independent VM instances in the same process) and compressed references (32-bit references to Java objects on 64-bit architectures) [18].
- **G1 GC**
  Multi-threaded GC for large heaps and multi-core machines that is optimized to reduce stop-the-world pauses, and improve latency while achieving high throughput.
- **Epsilon GC**
  No-op GC which never frees any allocated memory.

We will focus on the Serial GC in this thesis.

# 3 Fundamentals of the Substrate VM

This chapter discusses memory management aspects that are specific to GraalVM Native Image, including native memory access and the fundamentals of the heap architecture.

## 3.1 Native Memory Access from Java

GraalVM is written in Java, which has significant limitations on native memory access for many reasons including security risks and cross-platform compatibility. However, native memory access is a crucial necessity for the proper functioning of the GC.

The Substrate VM accesses native memory using interfaces provided by the GraalVM compiler. These interfaces allow us to work with native memory in a way that is quite similar to C. Most notably, there is the `Pointer` interface for accessing raw memory, it allows us to perform pointer arithmetic. The `Word` interface is also heavily used and represents word-sized values.

Listing 3.1 shows a simplified code snippet that uses the mentioned interfaces. It sets the marked state in the object header by applying predefined bit masks.

```
1 public static void setMarkedState(Object obj) {
2     Pointer ptr = Word.objectToUntrackedPointer(obj)
3     Word header = ptr.readWord(getHubOffset());
4     header = header.or(MARKED_OR_FORWARDED_BIT).or(REMEMBERED_SET_BIT);
5     ptr.writeWord(getHubOffset(), header);
6 }
```

**Listing 3.1:** Code snippet showing native memory access

## 3.2 Image Heap versus Java Heap

Native Image splits the heap into two parts [19] as visualized in Figure 3.1. This emerges from the application initialization and heap snapshotting at build time.

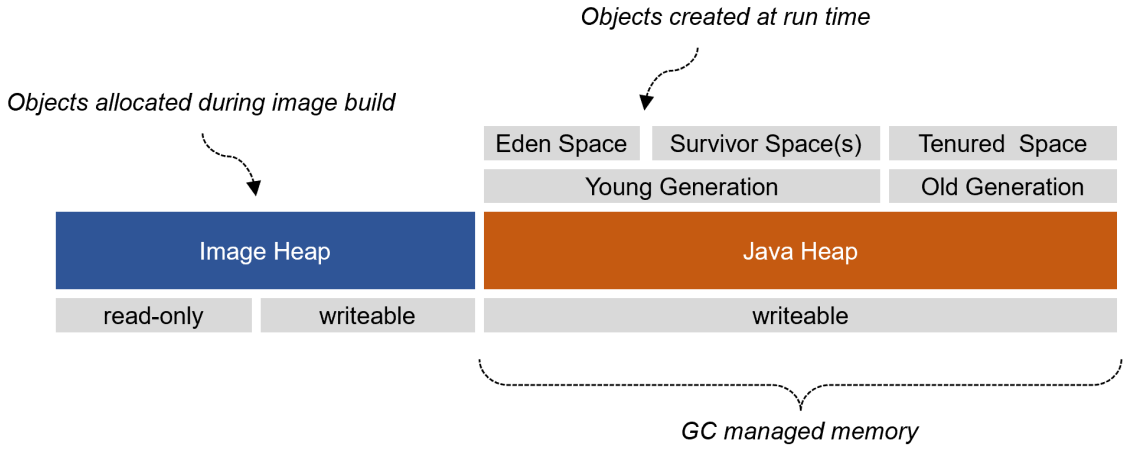



**Figure 3.1:** Differentiation of image heap and Java heap

At runtime, the so-called **image heap** contains objects created during the image build. It is pre-initialized with data from the data section of the executable binary and is available immediately upon application startup.
The objects in the image heap are immortal, i.e. the GC treats references in its objects as root pointers. However, the GC only has to scan parts of the image heap for root pointers. Objects that are identified as read-only during the image build and thus end up in the read-only section of the image heap can never have references to objects allocated at run time and therefore contain no root pointers. Similarly, objects that contain only primitive data and arrays of primitive types cannot contain root pointers and can thus be skipped by the GC.

On the other hand, the **Java heap** holds ordinary objects that are created at run time.
The GC reclaims memory occupied by dead objects in this part of the heap. The Java heap is built as a generational heap with aging. We will discuss this in more detail in Chapter 4.

## 3.3 Chunks

Chunks are the GC-internal unit for allocating and reclaiming memory. Both the image heap and Java heap, consist of multiple memory chunks instead of a continuous memory region (see Figure 3.2). There are two types of chunks:

- **Aligned Heap Chunks** (A)
  Aligned chunks are the common case and they can hold multiple objects. As the name suggests, these chunks are aligned to the heap base. This allows us to calculate the chunk address from any object pointer that points into this chunk by simply applying a bit-mask. The size of aligned chunks must be a power of two and can be adjusted using an option (`-H:AlignedHeapChunkSize`). It is set to 512KB by default.

- **Unaligned Heap Chunks** (U)
  Objects with a size above a certain threshold are considered large enough to be allocated in their own heap chunk. This optimization focuses especially on large arrays and prevents expensive copy operations during GC. The threshold can also be adjusted using an option (`-H:LargeArrayThreshold`) and is set to 128KB by default.

Figure 3.2 shows how chunks are organized as doubly-linked lists. Each chunk contains information about itself and adjacent chunks in its chunk header. This includes the offset to the previous and the next chunk of the same type.

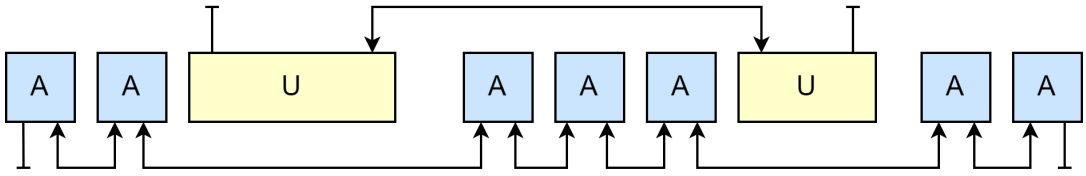



**Figure 3.2:** Doubly linked chunks

# 4 Our New Mark-and-Compact GC

This chapter focuses on the new compacting GC implemented in this thesis. We will break down its architecture, explain the sequential steps comprising a collection cycle, and discuss the different approaches taken during the implementation of this GC.

## 4.1 Architecture

As already mentioned in Chapter 1, our new GC is based on the existing GC implementation which provides both the Serial GC and the Epsilon GC in the Substrate VM. This decision was made due to the significant amount of effort required to implement a GC completely from scratch. In addition, many of the components would be implemented similarly, resulting in a lot of duplicated code.

To accomplish the new compacting GC, a feature flag has been introduced to alter the control flow of the existing implementation. This allows us to share most of the underlying source code while using a different algorithm for reclaiming memory. The value of the feature flag is set before image creation and cannot be changed afterward as it becomes a compile-time constant. The feature flag checks do not cause any runtime overhead thanks to the Graal compiler's optimizations such as constant folding, dead code elimination, and conditional elimination.

The core classes of the GC are shown as an UML diagram in Figure 4.1. Classes colored in white were pre-existing classes from the Serial GC that required modifications to incorporate the new GC while classes added in the process of this thesis are colored in green. The GC implementation makes heavy use of the visitor pattern to implement the algorithms it uses. As a result, we also implemented the phases of the Mark-and-Compact algorithm using the same design pattern.

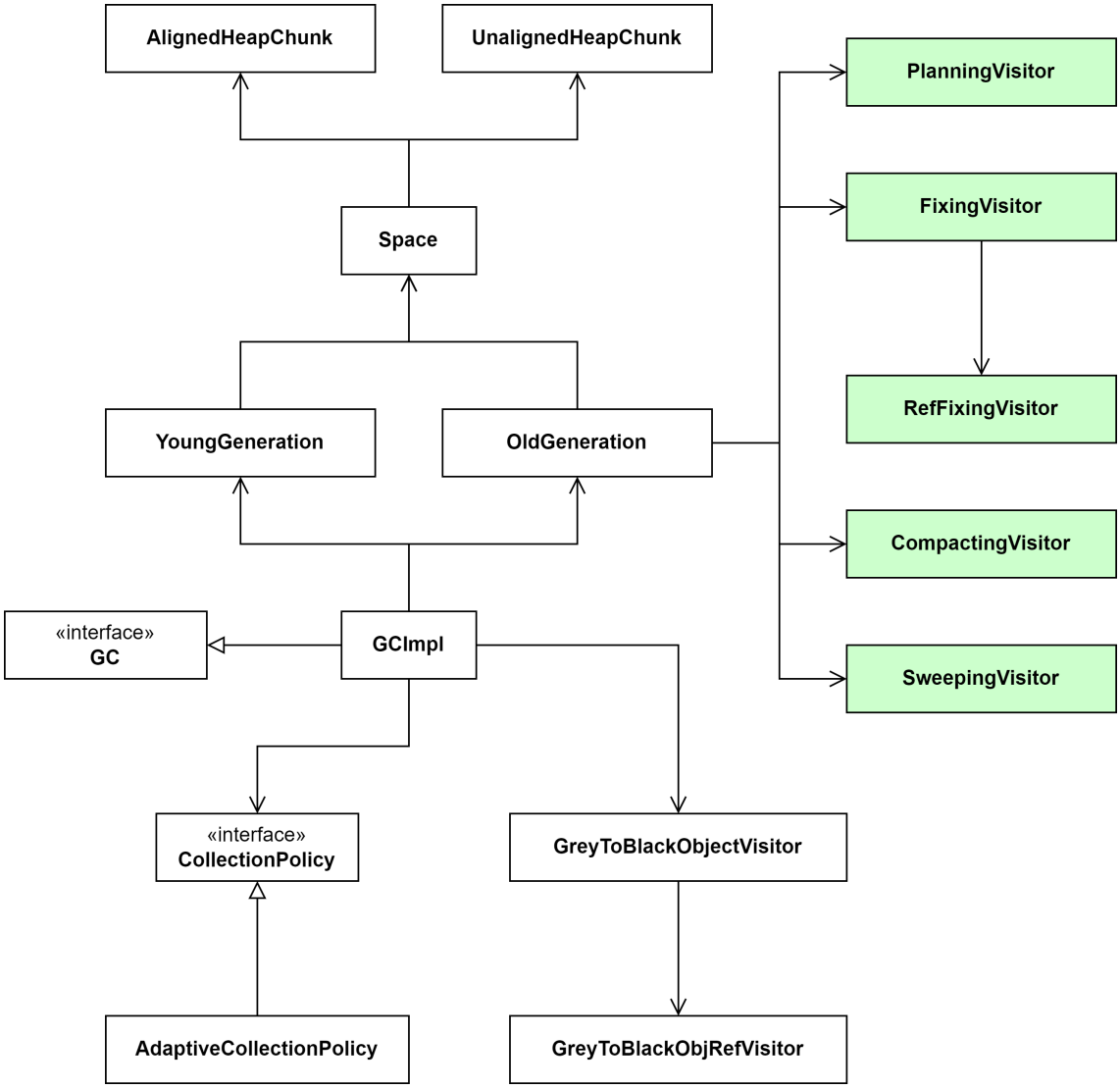



**Figure 4.1:** Core classes of the GC

**`Space`, `AlignedHeapChunk`, and `UnalignedHeapChunk`:**
A space is a collection of heap chunks, which hold the actual objects as described in the previous chapter. Heap chunks are doubly-linked. Thus, each space only holds references to the first and the last aligned chunk, as well as to the first and the last unaligned chunk. Furthermore, all three classes offer various utility methods.

**`YoungGeneration`, and `OldGeneration`:**
The GC uses a generational heap. The young generation holds the eden space for newly-allocated objects and the survivor spaces for short-living objects. The old generation has only the tenured space for promoted, long-living objects.

**`GC`, and `GCImpl`:**
The `GCImpl` class serves as the central component of the GC, connecting all other components. It houses most of the scavenging-related source code and provides verification utilities to make bug finding easier.

**`CollectionPolicy`, and `AdaptiveCollectionPolicy`:**
A collection policy defines the strategy and rules for reclaiming memory, including the frequency and timing of garbage collection. By default, the adaptive collection policy is being used, which aims to balance throughput and memory footprint. It uses various configurable parameters, cost estimators, and timers to dynamically adjust heap sizes and collection frequency based on the application's characteristics.

**`GreyToBlackObjectVisitor`, and `GreyToBlackObjectRefVisitor`:**
These classes are used to traverse the object graph, visiting each live object and its references. They implement both the full Stop-and-Copy algorithm used for minor collections and the mark phase of the Mark-and-Compact algorithm for the major collection cycles. We will go into more details in the subsequent sections of this chapter.

**`PlanningVisitor`, `FixingVisitor`, `RefFixingVisitor`, `CompactingVisitor`, and `SweepingVisitor`:**
These classes have been added to implement the remaining phases of the Mark-and-Compact algorithm. Contrary to the visitors used for marking, they sequentially iterate through the objects in a heap chunk. Further details will be provided in later sections of this chapter.

## 4.2 Generational Heap

The Java heap is built as a generational heap with aging. Objects are first allocated in the young generation, specifically in the eden space. Aging is being implemented by having multiple layers of survivor spaces (default: 16) as shown in Figure 4.2. The survivor spaces are also part of the young generation. Whenever a garbage collection cycle is triggered, surviving objects of the *Eden* space are copied to *Survivor-1* space and surviving objects of *Survivor-n* space are copied to *Survivor-n+1* space. Surviving objects from *Survivor-16* space are copied to *Tenured* space, which is part of the old generation.

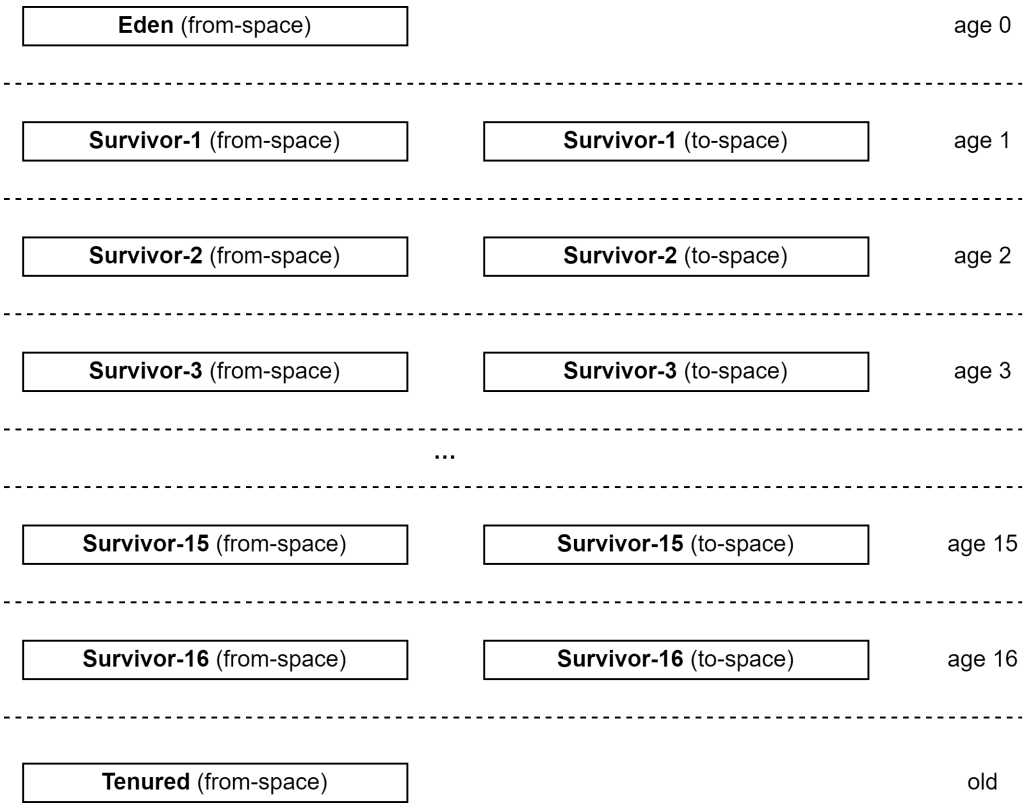



**Figure 4.2:** Generational heap

In addition, a major collection cycle promotes all reachable objects to the old generation. As already described in Chapter 1, our focus is on implementing the Mark-and-Compact algorithm for the old generation, i.e. for the tenured space.

## 4.3 Minor Collection

A so-called minor collection reclaims memory in the young generation only. Many objects in this generation have short lifetimes and become unreachable relatively quickly. This especially applies to the objects in the eden space. Therefore, it is preferable to perform garbage collection more frequently in the young generation than in the old generation.

> At the time of writing this thesis, a minor collection is called an *incremental* collection in the source code. This has caused some confusion when talking to colleagues as the GC does **not** split up the collection task into incremental steps. This applies to both the new compacting GC and the existing Serial GC.
> Thus, search for "`incremental`" in the `GCImpl` class in case you are interested in the actual source code and don't be fooled.

The goal is to speed up the identification and collection of these short-lived objects that are no longer reachable. This drastically improves the GC's performance in applications with a high object churn rate. The churn rate describes the number of allocations of temporary objects per transaction or time slice [20]. Java allows us to allocate and discard a large number of objects very quickly. Our GC has to keep up with that as well as possible while accepting that beyond a certain churn rate, more time is spend on garbage collection than on executing the actual program.

Assuming that the majority of objects in the young generation are dead at the time of a minor collection, we can only expect a minimal improvement in peak memory usage when switching from the Stop-and-Copy algorithm to the Mark-and-Compact algorithm. Therefore, it makes sense to keep the Stop-and-Copy algorithm for the young generation as the throughput and latency penalties of the Mark-and-Compact algorithm are not worth it.

Listing 4.1 provides a simplified abstract description of the algorithm used by the minor collection in our new GC. It mostly follows the original Stop-and-Copy algorithm described in 1970 by C. J. Cheney [7], also known as Cheney's algorithm. Note that additional steps that are related to advanced features of the GraalVM were left out for complexity reasons.

```
scavenge() =
    // Scan all root pointers
    foreach root_reference in roots:
        forwarding_address = copy(root_reference.to_object())
        root_reference.set(forwarding_address)

    // Scan objects forwarded to to-space
    scan_pointer = to_space.get_start()
    while scan_pointer < to_space.top:
        holder_object = scan_pointer.to_object()

        // Visit all outgoing references
        foreach reference in holder_object:
            forwarding_address = copy(reference.to_object())
            reference.set(forwarding_address)

        // Move on to the next object
        scan_pointer = scan_pointer + holder_object.size()

    swap(fromspace, tospace)

copy(object) =
    if object.has_forward_flag():
        return object.get_forwarding_address()

    // Copy to object contents forward and set forwarding pointer
    new_address = to_space.promote(object)
    object.set_forwarding_address(new_address)

    return new_address
```

**Listing 4.1:** Minor collection algorithm

## 4.4 Major Collection

A major collection cycle performs comprehensive memory reclamation across the entire Java heap, including both the young generation and the old generation. However, our main focus is on reclaiming unreachable long-lived objects in the tenured space to maintain an overall healthy and efficient memory management state.

Furthermore, all live objects are promoted to the old generation during major collections. Although this results in copying also newly allocated short-lived objects to the tenured space, it greatly reduces the complexity of GC implementation.

The new GC uses the Mark-and-Compact algorithm for the major collection. We assume that the majority of objects will be reachable, making the compacting approach the better fit. As shown in Figure 4.3, we can split the procedure into four phases.


**Figure 4.3:** Major collection phases

### 4.4.1 Mark Phase

The mark phase is the first step in the Mark-and-Compact algorithm. We start with the assumption that all objects in the Java heap are considered unreachable and are thus unmarked. This precondition is not checked by default due to the performance overhead that comes with it but it can be enabled together with further verification procedures using the option `-H:+VerifyHeap` at build time.

The GC identifies live objects in the Java heap in this phase by traversing the object graph, starting with the root pointers, which can be found in objects on the image heap, in references on the stack, in pinned objects, and in thread locals. If the image is built with the run-time compilation feature enabled, we will also find root pointers located in the run-time code cache.

Given a known-reachable object from the root pointers or the mark queue, the traversal routine performs the following steps:

- Verify that the given object resides in the Java heap and was not visited already and is thus not marked yet.
- If the object is pinned and thus must not be moved by the GC, set the should-sweep-instead-of-compact flag in the chunk header to `true`. Note that this step is skipped if the object is in an unaligned chunk.
- Mark the object by setting the marked state in the object header, i.e. setting both the mark-or-forward bit and the remembered-set bit to `true`.
- Add outgoing references held by the object's fields to the mark queue.

Traversal ends when there are no more objects to visit in the mark queue and thus all reachable objects in the Java heap have been marked as shown in Figure 4.4.

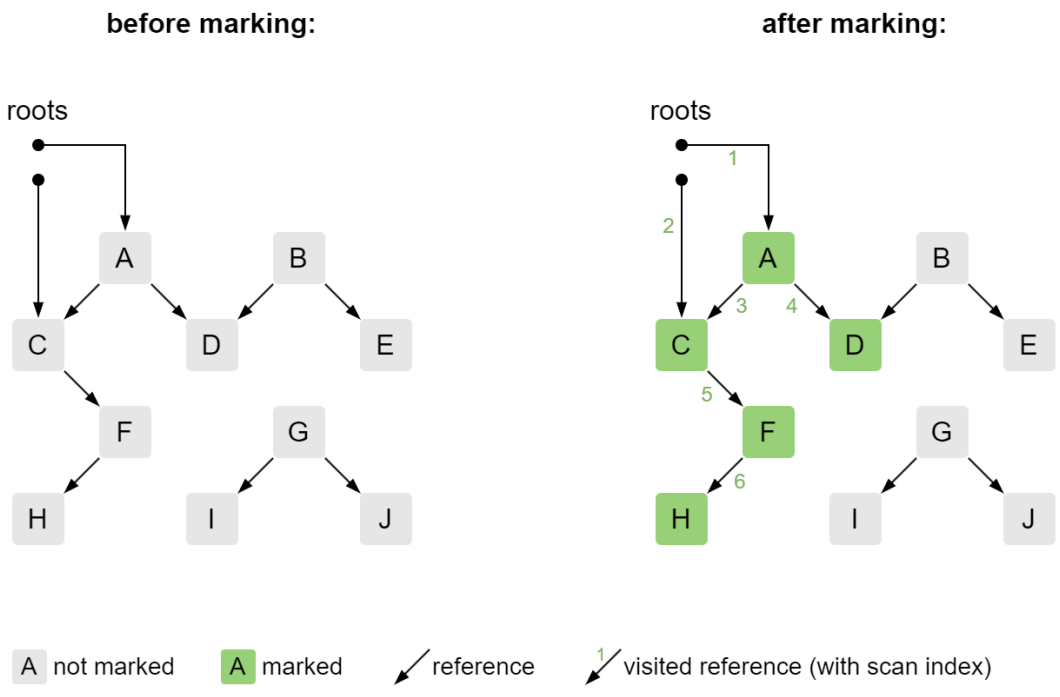



**Figure 4.4:** Marking of live objects

### Mark Queue

The mark queue is a data structure used to keep track of grey objects, i.e. it holds references to live objects, which still need to be visited during the mark phase. Entries are stored and retrieved in FIFO (first-in-first-out) order using `push(element)` and `pop()` operations.

> In the beginning, we used recursive method calls for the traversal of the object graph. However, this approach was not feasible as we quickly exceeded the limit of the call stack when encountering deep objects like `LinkedList`.
> The solution for this issue was to replace the recursive approach with an iterative one using off-heap memory.

The `MarkQueue` uses native memory to store references in segments, i.e. we are requesting memory directly from the operating system using `LibC.malloc(size)` as shown in Listing 4.2. A mark queue `Segment` is a raw structure that can store a fixed number of references similar to an array. As a result of directly requesting native memory, we have to take special care to release the allocated memory again using `LibC.free(ptr)` at the end of the mark phase.

```
1  @AlwaysInline("GC performance")
2  @Uninterruptible(
3      reason = CALLED_FROM_UNINTERRUPTIBLE_CODE,
4      mayBeInlined = true
5  )
6  private static Segment allocateSegment() {
7      UnsignedWord size = SizeOf.unsigned(Segment.class).add(
8              ENTRIES_PER_SEGMENT *
                   ConfigurationValues.getObjectLayout().getReferenceSize()
9      );
10     Segment segment = LibC.malloc(size);
11     segment.setNext(WordFactory.nullPointer());
12     return segment;
13 }
```

**Listing 4.2:** Allocation of a new segment in the mark queue.

### 4.4.2 Plan Phase

Now that the reachable objects have been identified and marked, the GC needs to plan the future layout of the heap after the compaction. Note that this phase only applies to aligned heap chunks, since only they are subject to fragmentation. Unaligned chunks cannot be compacted because they hold a single large object that will either remain in place if it has been marked, or the unaligned chunk will be released as a whole if its object has not been marked. Also note that compaction is performed across aligned chunks, meaning that live objects may be moved to a different chunk.

The main goal of the plan phase is to define the new location of all live objects. The GC is also faced with the decision of whether to compact or sweep individual aligned chunks based on their fragmentation. In the case of aligned chunks containing a pinned object, this decision has already been made in the mark phase, since pinned objects must not be moved during collection.

The GC performs three tasks in a single pass through all objects in aligned heap chunks:

- Group objects into plugs and gaps.
- Define relocation pointers.
- Build up the brick table.

> The related source code of this phase can be found in the `PlanningVisitor`, `RelocationInfo` and `BrickTable` classes.

Subsequent phases will execute the results of this phase straightforwardly, making the plan phase highly impactful and a key factor for the performance and efficiency of the GC.

### Plugs and Gaps

During the plan phase, all aligned chunks in the old generation are scanned object by object and grouped into plugs and gaps.

The concept of plugs and gaps is being used by the .NET Runtime [21] and we also decided to utilize this approach for our GC. The core principle is to group all objects based on whether they were marked in the previous phase as shown in Figure 4.5. This results in two kinds of groups:

- **Plug** representing a group of adjacent live objects.
- **Gap** representing a group of adjacent dead objects.

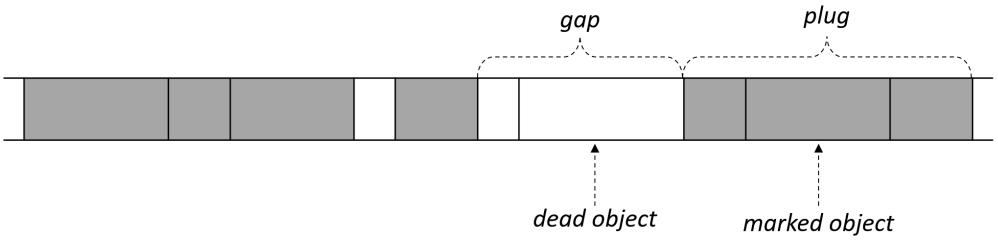



**Figure 4.5:** Plugs and gaps

This approach offers the advantage of being able to layout and move whole plugs during upcoming phases instead of working on an object-level. This improves the performance of the GC.

### Relocation Info

As mentioned earlier, the primary task in the plan phase is to define the new location of live objects after compaction. The GC performs this step on a per-plug basis and persists the information directly in front of the corresponding plug using the memory that is currently occupied by dead objects as depicted in Figure 4.6. As a result, we do not have to store the relocation pointers in the object headers.

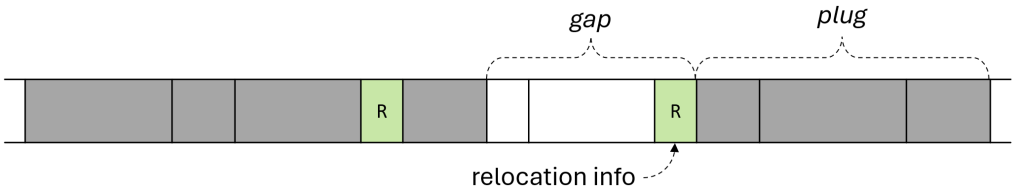



**Figure 4.6:** Relocation info stored in gaps

The relocation info structure includes the following fields:

- **Relocation pointer**
  The address (or the offset on EE) of the plug's new location after compaction.
- **Gap size**
  The size of the gap in which the relocation info resides.

  The GC requires both the gap size and the plug size during the final compaction phase. The plug size can be calculated using the *next offset* (see below) and the gap size from the succeeding relocation info.
- **Next offset**
  The distance between this plug and the next plug.

  The next offset links all relocation info structures in an aligned chunk, creating a singly-linked list. This allows us to iterate them sequentially, which will be necessary in upcoming phases.

As a consequence of storing the relocation information in gaps, the relocation info structure must fit into the smallest possible gap. Currently, the minimum object size defined by the size of `Integer` instances is 16 bytes on GraalVM CE Native Image. This allows us to use 8 bytes for the relocation pointer, 4 bytes for the gap size, and the remaining 4 bytes for the next plug offset.

However, the EE offers advanced optimizations which reduce the minimum object size to 8 bytes. This forces us to drastically reduce the size of the fields. In EE, the relocation pointer is therefore stored as a signed offset in 4 bytes. The gap size and the next plug offset fields are also reduced to 2 bytes each, which reduces their value range to the absolute minimum required.

### Plug Allocator

The relocation pointer is defined using a simple bump pointer strategy, setting the next relocation pointer to the old one plus the size of the corresponding plug. It preserves the relative placement of the live objects in the old generation. The objects maintain their sorting based on age, with those promoted to the old generation earlier positioned close to the first aligned chunk of the tenured space, while recently promoted objects are positioned near the last aligned chunk.

Figure 4.7 illustrates for a small example memory fragment how plugs are placed next to each other during compaction. At the start of the plan phase, the allocation pointer is positioned at the beginning of the first aligned chunk in the tenured space. Once the GC is at the end of a plug and it is time to write the relocation information into the preceding gap, it gets the relocation pointer from the current value of the allocation pointer. Finally, the plug size is added to the allocation pointer.

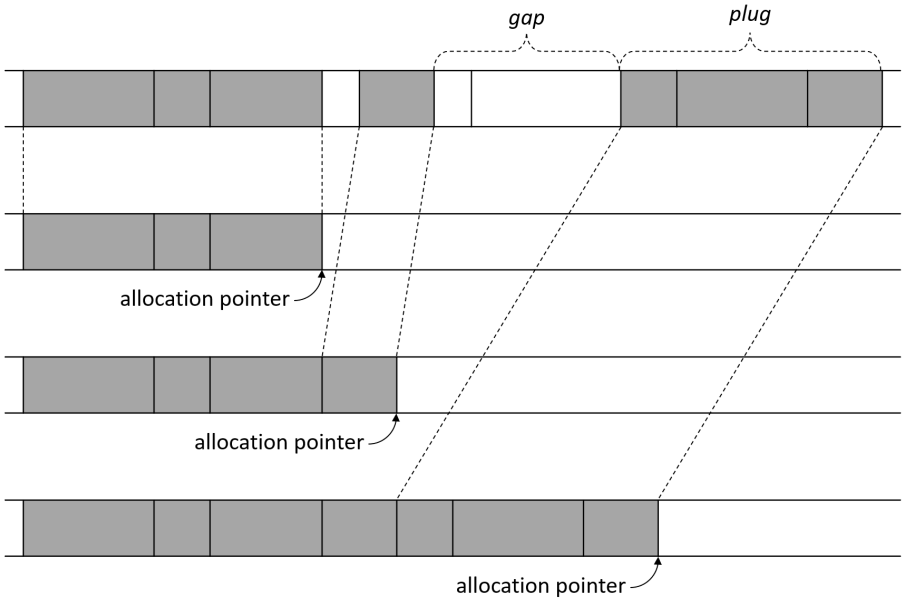



**Figure 4.7:** Example for plug allocation on chunk compaction

However, the allocator's behavior changes if the given chunk needs to be swept instead of compacted. There are two situations in which this is the case:

- The should-sweep-instead-of-compact flag in the chunk header is set to `true` because a pinned object is located in the chunk, which prevents plugs from being moved.
- The given chunk has low internal fragmentation, meaning that gaps account for less than 6.25% of the chunk.

  The GC keeps track of the total gap size in a chunk while defining the relocation pointer using the bump pointer strategy. When the GC reaches the end of the chunk, it calculates the fragmentation ratio by dividing the total gap size by the chunk size. If the ratio is below the 6.25% threshold, it adjusts all relocation pointers in the chunk for sweeping, sets the should-sweep-instead-of-compact flag to `true`, and sets the bump pointer to the end of the last plug in the chunk.

The relocation pointer is set to the plug's current position as shown in Figure 4.8. The plugs will remain at their current location and will not be moved during compaction.

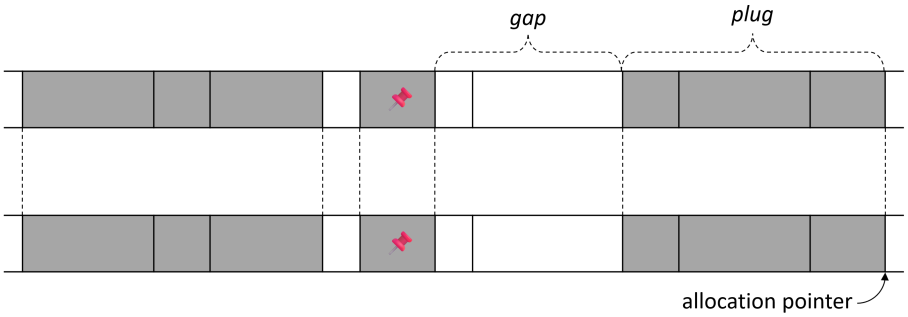



**Figure 4.8:** Example for plug allocation on chunk sweep

### Brick Table

The task of retrieving the relocation pointer for a given object can be very costly, because it is stored in front of the plug containing this object. The GC needs to find the start of this plug. This can only be done by iterating over all plugs starting from the first plug of the enclosing aligned chunk. Aligned chunks may contain thousands of plugs which makes it very expensive given how often the GC has to perform this task and thus impractical.

In order to overcome this issue, we again took a look at the .NET Runtime and implemented a similar solution to their brick table concept [21]. The brick table acts as an index to shortcut the iteration over all plugs in the enclosing chunk as depicted in Figure 4.9. It provides a pointer to a preceding plug near the given object that can be used as a starting point for iteration instead of starting from the first plug of the chunk.

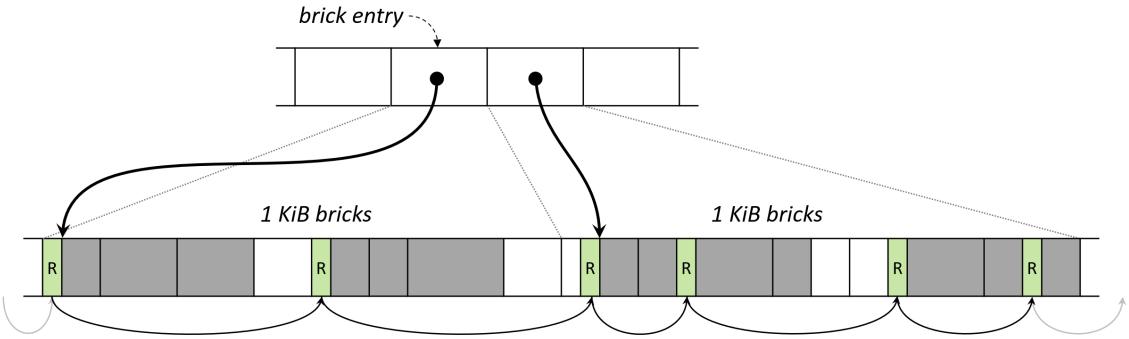



**Figure 4.9:** Brick table

Our approach divides chunks into one kilobyte ranges (so-called *bricks*) aligned to the heap base. Each entry in the brick table covers one brick and stores a 16-bit unsigned integer value which represents the offset of the first plug in this brick relative to the chunk start. To find the relocation offset of an object, the object's address is simply divided by 1024 to get the index into the brick table and thus the first plug in this brick. From there, the plug containing the object can easily be found by following the next pointers of the plugs. The brick table itself is stored in the aligned chunk header, temporarily using the memory of the card table.

### 4.4.3 Reference Fixup Phase

It is finally time to fix up all object references as a last step before compaction. As most objects will be moved, references to these objects need to be updated to reflect their future memory locations. While most of the preparation work has already been done in the plan phase, the reference fixup phase takes the longest time compared to the other phases.

The GC sequentially revisits all roots and marked objects. It performs the following steps for every visited object reference:

1. Skip the reference if it points to an object in either the image heap or an unaligned chunk as in both cases the referenced object will not be moved during compaction.
2. Retrieve the relocation pointer of the referenced object using the brick table.
3. Update the reference based on the calculated new location.
4. If the holder object resides in the image heap, then dirty its card table entry.

> The related source code of this phase can be found in the `FixingVisitor` and `RefFixingVisitor` classes. Note that the reference map of `Reference<T>` subclasses does not include their `referent` field and thus must be visited explicitly.

At the end of this phase, the GC has updated all object references and simultaneously also the card table of the image heap.

While the GC is now finally ready to compact the old generation, we have to take special care to not access any of the updated references as objects have not been moved yet. This may seem like a trivial restriction to mention, but it is one that can easily be overlooked in the context of reporting GC statistics.

> During the implementation of our GC, we faced issues with reporting JFR events that were used to track the duration of individual phases. As it turns out after debugging, the Substrate VM's JFR support also uses objects held in the Java heap.

### 4.4.4 Compaction Phase

Finally, the GC is ready to reclaim the memory occupied by dead objects by executing the decisions it made earlier. Depending on the state of the should-sweep-instead-of-compact flag in the aligned chunk header, the GC either compacts or sweeps the given chunk.

In case the flag has not been set to `true` and thus the given aligned chunk can be compacted, the GC iterates over all plugs and performs the following steps:

1. Retrieve the relocation pointer of the plug
   and skip the plug if it matches the current location.
2. Calculate the plug size and skip the plug if its size equals zero.
3. Copy the whole plug to its new location.
4. Update the top pointer of the destination chunk.

However, if it was decided in an earlier phase that a given aligned chunk should be swept, the GC instead iterates over all gaps and performs these steps:

1. Retrieve the gap size and skip the gap if its size equals zero.
2. Generate a `byte[]` filler object of equal size.
3. Write the generated object header to the start of the gap.

> The related source code of this phase can be found in the `CompactingVisitor` and `SweepingVisitor` classes.

After compaction, the old generation now holds empty aligned chunks that the GC can finally release. At this point, the GC also revisits the unaligned chunks and either releases the chunk if it contains an unmarked object or clears the marked state of the contained object if it has been marked. While this completes the Mark-and-Compact algorithm, the GC still has to perform a few cleanup tasks, such as cleaning the runtime code cache, before returning to the application. The GC can collect the generated filler objects on its next run if it turns out that the affected chunks can be compacted.

# 5 Performance Evaluation

In this chapter, we dive into the performance evaluation of our new compacting GC and compare its results to the existing Serial GC. Our focus is to verify that the new compacting GC provides a reduced memory footprint while still maintaining a high throughput. We describe the benchmarks and methodology used to do this, before finally presenting the obtained results.

## 5.1 Benchmarks

The main aspects for evaluating the performance of a GC are throughput, latency, and footprint [17]:

- **Throughput** is the percentage of total time not spent in GC considered over long periods of time. The throughput of the GC greatly affects the throughput of the entire application. Thus, we can measure this aspect indirectly by measuring how many operations an application can handle.
- **Latency** is the responsiveness of an application. Especially long GC pauses have a negative effect on responsiveness. However, we are not focusing on this aspect since our new compacting GC and the existing Serial GC are not optimized for short GC pauses.
- **Footprint** is the working set size of a process measured in the number of bytes. We are particularly interested in the maximum memory usage, as we are hoping for a noticeable improvement in this aspect.

The GraalVM repository and its build tool provide several benchmark suites. The selected benchmark scenarios for this thesis are based on two microservice frameworks as this matches the target use-case of GraalVM Native Image as described in Section 2.2.1.

- **Spring PetClinic**
  Spring [22] is a comprehensive Java framework that promotes modular and scalable applications with a convention-over-configuration philosophy. We use the very popular PetClinic [23] sample application for our benchmarks.
- **Micronaut ShopCart**
  Micronaut [24] is a lightweight and fast Java framework designed for building microservices and serverless applications, emphasizing minimal runtime reflection and quick startup times. For our benchmarks, we use a demo application called ShopCart, which provides similar APIs to PetClinic, but in a different domain.

Both microservice applications provide a REST API for interaction, which the benchmarks use to generate the workload by executing various requests. After applying a constant load to warm up the application, the benchmarks take their measurements.

The benchmarking experiment ran on dedicated Oracle X5–2 servers in the Oracle Cloud:

- **CPU:** Intel Xeon E5–2699 v3
- **Memory:** 72 GB DDR4
- **Operating System:** Oracle Linux 8.8

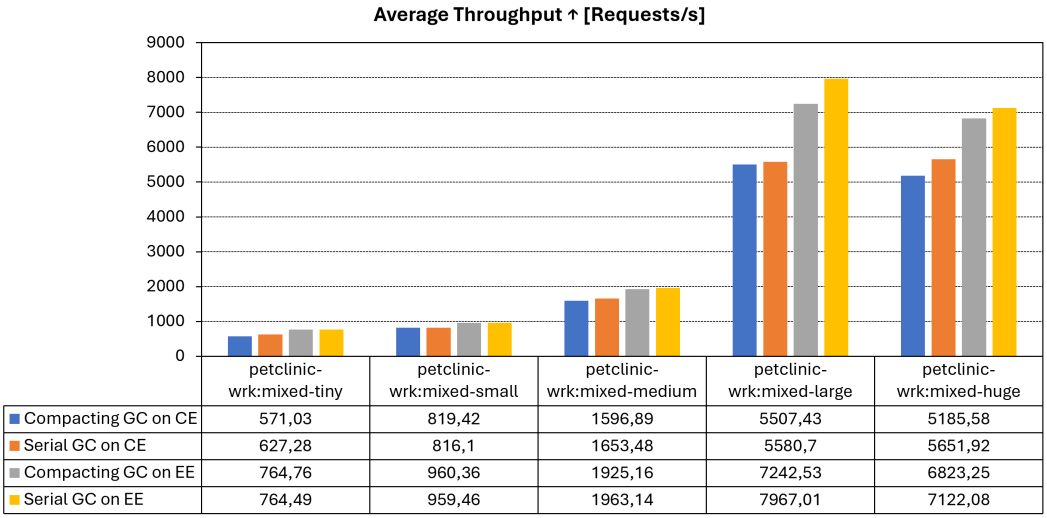
To better simulate real-world deployments, the benchmarks were conducted with specific compute and memory resource limits on the microservice applications we are testing as listed in Table 5.1. The initial Java heap size and the maximum Java heap size are limited using the `-Xms` and `-Xmx` JVM parameters. The number of available CPU cores is limited using the `-XX:ActiveProcessorCount` JVM parameter and a CPU binding using the `hwloc-bind` command.

| Scenario name | Initial Java heap size (MB) | Maximum Java heap size (MB) | Number of available CPU cores |
|---|---|---|---|
| petclinic-wrk:mixed-tiny | 32 | 100 | 1 |
| petclinic-wrk:mixed-small | 40 | 144 | 2 |
| petclinic-wrk:mixed-medium | 80 | 256 | 4 |
| petclinic-wrk:mixed-large | 320 | 1280 | 16 |
| petclinic-wrk:mixed-huge | 640 | 3072 | 32 |
| shopcart-wrk:mixed-tiny | 32 | 112 | 1 |
| shopcart-wrk:mixed-small | 64 | 224 | 2 |
| shopcart-wrk:mixed-medium | 128 | 512 | 4 |
| shopcart-wrk:mixed-large | 512 | 3072 | 16 |
| shopcart-wrk:mixed-huge | 1024 | 8192 | 32 |

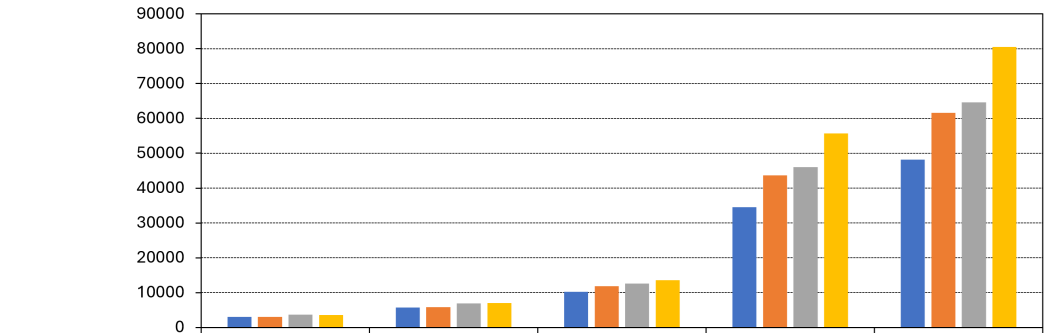
**Table 5.1:** JVM parameters used when starting the microservice application for the benchmark scenarios.

## 5.2 Results

Figure 5.1 shows the obtained throughput results. Our new compacting GC loses 6.9% on average across both benchmarks and all workloads compared to the existing Serial GC. This is notably true for the Micronaut Shopcart benchmarks where a reduction of up to 22% can be observed. However, these results are not surprising because the compacting approach requires additional passes through the heap and thus causes longer pauses.



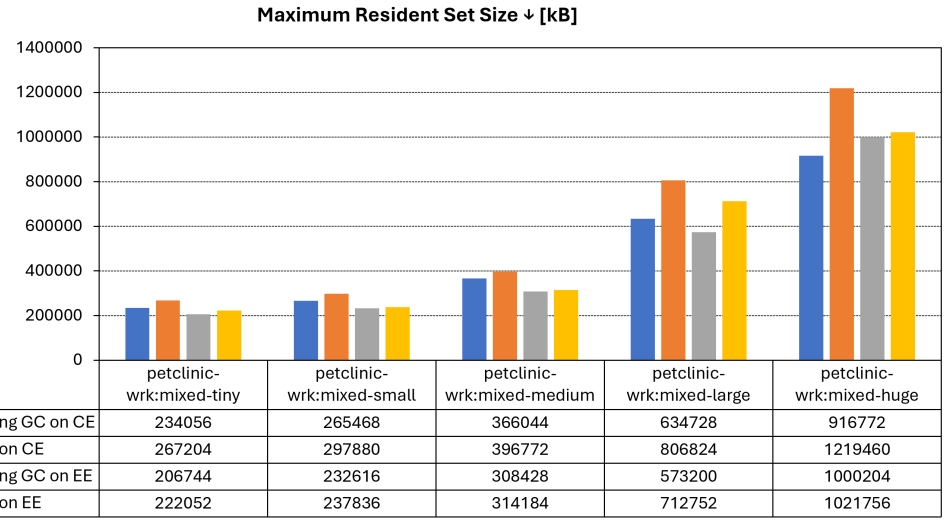

| | petclinic-wrk:mixed-tiny | petclinic-wrk:mixed-small | petclinic-wrk:mixed-medium | petclinic-wrk:mixed-large | petclinic-wrk:mixed-huge |
|---|---|---|---|---|---|
| Compacting GC on CE | 571,03 | 819,42 | 1596,89 | 5507,43 | 5185,58 |
| Serial GC on CE | 627,28 | 816,1 | 1653,48 | 5580,7 | 5651,92 |
| Compacting GC on EE | 764,76 | 960,36 | 1925,16 | 7242,53 | 6823,25 |
| Serial GC on EE | 764,49 | 959,46 | 1963,14 | 7967,01 | 7122,08 |



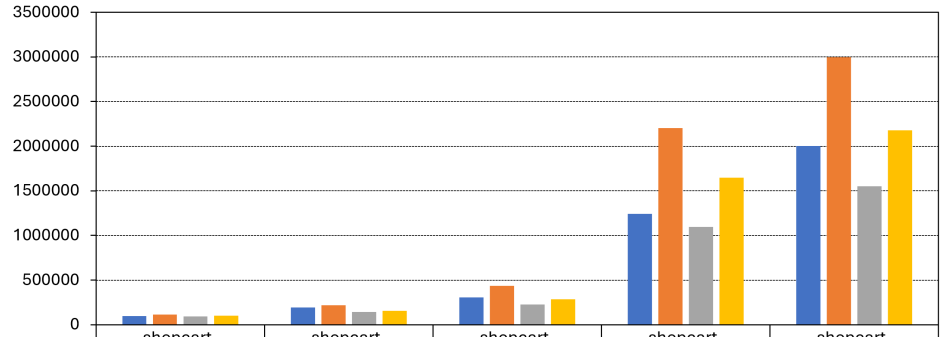

| | shopcart-wrk:mixed-tiny | shopcart-wrk:mixed-small | shopcart-wrk:mixed-medium | shopcart-wrk:mixed-large | shopcart-wrk:mixed-huge |
|---|---|---|---|---|---|
| Compacting GC on CE | 3062,32 | 5771,27 | 10217,55 | 34530,27 | 48170,46 |
| Serial GC on CE | 3048,96 | 5813,51 | 11814,93 | 43638,7 | 61549,47 |
| Compacting GC on EE | 3638,31 | 6888,99 | 12626,43 | 46016,98 | 64585,88 |
| Serial GC on EE | 3592,24 | 7032,18 | 13570,58 | 55683,38 | 80499,75 |

**Figure 5.1:** Average throughput results

Figure 5.2 shows the obtained maximum memory footprint results. Our new compacting GC reduces the maximum resident set size (RSS) of the selected microservice applications by 21.1% on CE and 12.9% on EE on average across both benchmarks and all scenarios. The CE of GraalVM Native Image seems to benefit the most from the new compacting GC. Furthermore, all scenarios that suffered a decrease in average throughput also achieved an even greater improvement in maximum memory usage. The *shopcart-wrk:mixed-large* scenario showed the greatest relative improvement of 43.8% on CE and 33.6% on EE.




| | petclinic-wrk:mixed-tiny | petclinic-wrk:mixed-small | petclinic-wrk:mixed-medium | petclinic-wrk:mixed-large | petclinic-wrk:mixed-huge |
|---|---|---|---|---|---|
| Compacting GC on CE | 234056 | 265468 | 366044 | 634728 | 916772 |
| Serial GC on CE | 267204 | 297880 | 396772 | 806824 | 1219460 |
| Compacting GC on EE | 206744 | 232616 | 308428 | 573200 | 1000204 |
| Serial GC on EE | 222052 | 237836 | 314184 | 712752 | 1021756 |




| | shopcart-wrk:mixed-tiny | shopcart-wrk:mixed-small | shopcart-wrk:mixed-medium | shopcart-wrk:mixed-large | shopcart-wrk:mixed-huge |
|---|---|---|---|---|---|
| Compacting GC on CE | 97748 | 191640 | 308284 | 1242492 | 2002068 |
| Serial GC on CE | 116096 | 218292 | 434480 | 2203400 | 2995756 |
| Compacting GC on EE | 94648 | 144352 | 226752 | 1094132 | 1549396 |
| Serial GC on EE | 100176 | 156876 | 283816 | 1647964 | 2176876 |

**Figure 5.2:** Maximum RSS results

# 6 Related Work

While Section 2.1 already presented the historic work related to this thesis, this chapter provides a brief overview of the algorithms used by today's state-of-the-art GCs.

## 6.1 .NET Runtime

.NET is an open-source, cross-platform developer platform maintained by Microsoft for building many different types of applications [25] similar to Java. It can run programs written in various languages, with C# being the most popular, and relies on a high-performance runtime that is used in production by many high-scale apps [26].

The .NET runtime divides the managed heap into two segments for managing small and large objects, the small object heap (SOH) and the large object heap (LOH). While the GC uses a Mark-and-Compact algorithm for the SOH, it uses a Mark-and-Sweep algorithm for the LOH as well as a free list to allocate new large objects [21].

## 6.2 HotSpot JVM

HotSpot is a widely used VM for the Java platform and is part of the OpenJDK project [27]. It provides multiple GC implementations to manage memory efficiently in various scenarios, including Serial GC, Parallel GC, and G1 GC.

The HotSpot's Serial GC uses the Mark-Sweep-Compact algorithm for collecting dead objects in both the young generation and the old generation according to its source code [28]. However, it seems that this was not always the case, as some web sources [29][30][31] still claim that it is copying collector using the Stop-and-Copy algorithm.

# 7 Conclusion

This chapter concludes this thesis by providing a summary of the work and an outline of possible future work to be done in this area.

## 7.1 Summary

This thesis presented a new compacting GC for GraalVM Native Image that is based on the existing implementation of its Serial GC. Using established best practices, we focused on minimizing the memory footprint while maintaining high throughput. The implemented solution makes use of the Mark-and-Compact algorithm for collecting unreachable objects during major collection cycles. In addition, it takes advantage of the heap being separated into chunks by selectively compacting chunks that contain a significant number of dead objects to reduce costly memory writes.

Furthermore, we evaluated the performance of our implementation. The new GC demonstrated a significant improvement in maximum memory usage compared to the existing Serial GC thanks to the compacting approach. However, we also observed a noticeable reduction in average throughput caused by longer GC pauses. This was expected due to the nature of the Mark-and-Compact algorithm requiring additional passes through the managed heap compared to the previously used Stop-and-Copy algorithm.

Moreover, this work contributed to the open-source community by providing the source code of our new GC to the official public GraalVM repository on GitHub [3]. This contribution is expected to benefit applications with tight memory constraints, such as microservices and function-as-a-service applications running in containerized environments.

## 7.2 Future Work

Firstly, the pull request with all our source code changes is currently being reviewed by the GraalVM team and is expected to be included in the release of GraalVM version 24.1.0 in September 2024 [32].

Although we have successfully achieved the goal of this thesis by reducing the memory footprint of the GC, we believe that there is still room for further optimization.

- **Locality-aware GC optimizations**
  The current GC does not take object locality into account, which results in objects that reference each other being scattered throughout memory and causing poor CPU caching behavior. Good object locality means that frequently accessed and related objects are located close to each other in memory. This optimizes performance by allowing efficient use of hardware caches and prefetching mechanisms, thus reducing memory access times [33][34].

- **Compacting of aligned chunks that contain pinned objects**
  While the number of pinned objects is expected to be rather small in most applications, we could further improve the memory footprint by pinning the individual plugs that contain a pinned object. Using a more sophisticated algorithm to define the future heap layout in the plan phase, the GC could compact the memory space in between pinned plugs and either fill up the resulting gaps with plugs from other chunks or create a free list for new object allocations.

- **Separating primitive objects in Java heap**
  In the context of GraalVM Native Image, primitive objects refer to boxed primitives, primitive arrays, and instances of classes that contain only primitive fields. By separating them, similar to what is done in the image heap, the GC could skip them during the reference fixup phase because they do not hold any object references. This could result in shorter GC pauses.

# List of Figures


2.1 Stop-and-Copy algorithm . . . . . 10
2.2 Mark-and-Compact algorithm . . . . . 12
2.3 GraalVM architecture [11] . . . . . 13
2.4 Native Image [14] . . . . . 14
2.5 Key advantages of AOT compilation [15] . . . . . 15

3.1 Differentiation of image heap and Java heap . . . . . 18
3.2 Doubly linked chunks . . . . . 19

4.1 Core classes of the GC . . . . . 21
4.2 Generational heap . . . . . 23
4.3 Major collection phases . . . . . 26
4.4 Marking of live objects . . . . . 27
4.5 Plugs and gaps . . . . . 30
4.6 Relocation info stored in gaps . . . . . 31
4.7 Example for plug allocation on chunk compaction . . . . . 32
4.8 Example for plug allocation on chunk sweep . . . . . 33
4.9 Brick table . . . . . 34

5.1 Average throughput results . . . . . 40
5.2 Maximum RSS results . . . . . 41

# List of Tables

5.1 JVM parameters used when starting the microservice application for the benchmark scenarios. . . . . 39

# Bibliography


[1] *GraalVM Native Image*. URL: https://www.graalvm.org/latest/reference-manual/native-image/ (visited on 2024-01-26) (cit. on pp. 5, 15).

[2] Eliot Moss Richard Jones Antony Hosking. *The Garbage Collection Handbook*. CRC Press, 2023. ISBN: 9781032218038 (cit. on pp. 5, 6, 8).

[3] *GraalVM repository*. URL: https://github.com/oracle/graal (visited on 2023-10-14) (cit. on pp. 6, 43).

[4] John McCarthy. "Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I". In: *Commun. ACM* (1960). ISSN: 0001-0782. DOI: 10.1145/367177.367199. URL: https://doi.org/10.1145/367177.367199 (cit. on p. 8).

[5] Rafael Lins Richard Jones. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. 1996. ISBN: 9780471941484 (cit. on p. 8).

[6] Robert R. Fenichel and Jerome C. Yochelson. "A LISP Garbage-Collector for Virtual-Memory Computer Systems". In: *Commun. ACM* 12.11 (Nov. 1969), pp. 611–612. ISSN: 0001-0782. DOI: 10.1145/363269.363280. URL: https://doi.org/10.1145/363269.363280 (cit. on p. 9).

[7] C. J. Cheney. "A Nonrecursive List Compacting Algorithm". In: *Commun. ACM* 13.11 (1970). ISSN: 0001-0782. DOI: 10.1145/362790.362798. URL: https://doi.org/10.1145/362790.362798 (cit. on pp. 9, 24).

[8] *GraalVM Homepage*. URL: https://www.graalvm.org/ (visited on 2024-01-26) (cit. on p. 13).

[9] *Graal Compiler*. URL: https://www.graalvm.org/latest/reference-manual/java/compiler/ (visited on 2024-01-26) (cit. on p. 13).

[10] *Truffle Language Implementation Framework*. URL: https://www.graalvm.org/latest/graalvm-as-a-platform/language-implementation-framework/ (visited on 2024-01-31) (cit. on p. 13).

[11] *GraalVM Introduction*. URL: https://www.graalvm.org/22.3/docs/introduction/ (visited on 2024-01-26) (cit. on p. 13).

[12] *GraalVM Native Image*. URL: https://www.graalvm.org/22.0/reference-manual/native-image/ (visited on 2023-10-15) (cit. on p. 14).

[13] *Substrate VM in Graal repository*. URL: https://github.com/oracle/graal/tree/master/substratevm (visited on 2024-01-31) (cit. on p. 14).

[14] *Updates on Class Initialization in GraalVM Native Image Generation*. URL: https://medium.com/graalvm/updates-on-class-initialization-in-graalvm-native-image-generation-c61faca461f7 (visited on 2023-12-31) (cit. on p. 14).

[15] *Twitter Post by Thomas Würthinger about AOT vs JIT*. URL: https://twitter.com/thomaswue/status/1145603781108928513 (visited on 2023-10-15) (cit. on p. 15).

[16] *Spring Boot: GraalVM Native Image Support*. URL: https://docs.spring.io/spring-boot/docs/current/reference/html/native-image.html (visited on 2023-10-15) (cit. on p. 15).

[17] *GraalVM Native Image: Memory Management*. URL: https://www.graalvm.org/latest/reference-manual/native-image/optimizations-and-performance/MemoryManagement/ (visited on 2023-10-29) (cit. on pp. 16, 37).

[18] Christian Wimmer. *Isolates and Compressed References: More Flexible and Efficient Memory Management via GraalVM*. URL: https://medium.com/graalvm/isolates-and-compressed-references-more-flexible-and-efficient-memory-management-for-graalvm-a044cc50b67e (visited on 2023-10-29) (cit. on p. 16).

[19] Christian Wimmer et al. "Initialize Once, Start Fast: Application Initialization at Build Time". In: *Proc. ACM Program. Lang.* 3.OOPSLA (Oct. 2019). DOI: 10.1145/3360610. URL: https://doi.org/10.1145/3360610 (cit. on p. 18).

[20] Alois Reitbauer et al. *Java enterprise performance: Out-Of-Memory Errors*. URL: https://www.dynatrace.com/resources/ebooks/javabook/other-java-memory-issues/ (visited on 2024-01-04) (cit. on p. 24).

[21] Konrad Kokosa. *Pro .NET Memory Management*. Apress, 2018. ISBN: 9781484240267 (cit. on pp. 30, 34, 42).

[22] *Spring Homepage*. URL: https://spring.io/ (visited on 2024-01-23) (cit. on p. 38).

[23] *Spring PetClinic Sample Application*. URL: https://github.com/spring-projects/spring-petclinic (visited on 2024-01-23) (cit. on p. 38).

[24] *Micronaut Homepage*. URL: https://micronaut.io/ (visited on 2024-01-23) (cit. on p. 38).

[25] *.NET documentation*. URL: https://learn.microsoft.com/en-us/dotnet/core (visited on 2024-01-27) (cit. on p. 42).

[26] *Introduction to .NET*. URL: https://learn.microsoft.com/en-us/dotnet/core/introduction (visited on 2024-01-27) (cit. on p. 42).

[27] *OpenJDK*. URL: https://openjdk.org/ (visited on 2024-01-27) (cit. on p. 42).

[28] *HotSpot Source Code*. URL: https://github.com/openjdk/jdk/tree/master/src/hotspot (visited on 2024-01-27) (cit. on p. 42).

[29] *Further Look At JVM Garbage Collection*. URL: https://www.callibrity.com/blog/further-look-at-jvm-garbage-collection (visited on 2024-01-31) (cit. on p. 42).

[30] *JVM Troubleshooting MOOC: Lesson 1*. URL: https://www.oracle.com/webfolder/technetwork/tutorials/mooc/JVM_Troubleshooting/week1/lesson1.pdf (visited on 2024-01-31) (cit. on p. 42).

[31] *HotSpot Storage Management*. URL: https://openjdk.org/groups/hotspot/docs/StorageManagement.html (visited on 2024-01-27) (cit. on p. 42).

[32] *GraalVM Release Calendar*. URL: https://www.graalvm.org/release-calendar/ (visited on 2024-01-28) (cit. on p. 44).

[33] Albert Mingkun Yang, Erik Österlund, and Tobias Wrigstad. “Improving program locality in the GC using hotness”. In: *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2020. London, UK: Association for Computing Machinery, 2020, pp. 301–313. ISBN: 9781450376136. DOI: 10.1145/3385412.3385977. URL: https://doi.org/10.1145/3385412.3385977 (cit. on p. 44).

[34] Duarte Patrício et al. “Locality-Aware GC Optimisations for Big Data Workloads”. In: Oct. 2017, pp. 50–67. ISBN: 978-3-319-69458-0. DOI: 10.1007/978-3-319-69459-7_4 (cit. on p. 44).