Runtime Object Lifetime Profiler for the Virtual Machine GraalVM

Exending Graal's Native Image Runtime SubstrateVM with Object Pretenuring

Domantas Lionas

Thesis submitted for the degree of
Master in Programming and System Architecture
60 credits

Department of Informatics
Faculty of mathematics and natural sciences

UNIVERSITY OF OSLO

Spring 2021

Runtime Object Lifetime Profiler for the Virtual Machine GraalVM

Exending Graal's Native Image Runtime SubstrateVM with Object Pretenuring

Domantas Lionas

Runtime Object Lifetime Profiler for the Virtual Machine GraalVM

http://www.duo.uio.no/

Printed: Reprosentralen, University of Oslo

Abstract

Microservices have become a popular architecture style for building large scale distributed systems. Compared to the monolithic approach, they offer several benefits. Some of these are, e.g., being loosely coupled, technologically flexible, and comparably easier to scale. However, one major problem with JVM-based microservices is the lengthy warm-up times. *GraalVM Native Image* tries to solve this problem through efficient AOT-compilation of JVM-based applications. However, being still a rather new tool within the Java-ecosystem, it has a lot room for improvement. To further improve upon the latency and throughput of microservice-based applications, we show how the *Runtime Object Lifetime Profiler* (ROLP), a profiler tracking the lifetimes of frequently allocated objects, can be used in combination with the GraalVM Native Image. Throughout the thesis, we present the functionality of ROLP and how it manages to improve the dynamic memory management decisions for the Native Image runtime.

Acknowledgements

I would like to thank my supervisors, Prof. Paulo Ferreira and Dr. Rodrigo Bruno, for their utmost patience and guidance throughout this thesis. Your availability and expertise has been vital for the both the implementation and the research done in this work.

Furthermore, I would like to thank all of my friends at the Department of Informatics. Without you I would definitely not be as adequate of a programmer as I am right now. All of the conversations, evenings at Bliss and Escape and trips to Dana's bakery will be remembered fondly.

Thank you to my girlfriend for all of the love and the delicious meals you have provided me with throughout this period.

Finally, I would like to thank my family for their moral support. Thank you for pushing me to pursue the path of computer science and always being there for me.

Domantas Lionas
Oslo, May, 2021

Contents

List of Acronyms

ROLP Runtime Object Lifetime Profiler

OLD Table Object Lifetime Distribution Table

GC Garbage collection

AST Abstract syntax tree

TLAB Thread-local allocation buffer

TSS Thread Stack State

VM Virtual Machine

JVM Java Virtual Machine

JIT compilation Just-in-time compilation

AOT compilation Ahead-of-time compilation

IGV Ideal Graph Visualizer

SOA Service-oriented architecture

SLA Service-level agreement

PGO Profile-guided optimization

EE Enterprise Edition

FOT First Object Table

List of Figures

List of Tables

List of Listings

Part I

Introduction, Background and Related Works.

Chapter 1

Introduction

Nowadays, the use of microservices is popular for building large scale distributed systems [33]. Compared to the monolithic approach, they offer several benefits. Some of these are, e.g., being loosely coupled, technologically flexible, and comparably easier to scale. Nevertheless, being an architecture style consisting of lightweight communicating services, there are problems which need to be addressed. For JVM-based microservices, those issues mainly fall under memory footprint and lengthy warm-up times [22, 39].

Since many microservice-based applications are built using JVMs, they are prone to spend a lot of time in the warm-up phase of the JVM [39]. The warm-up phase of a JVM-based application is the time between the startup (execution speed here is low), and until the steady peak of performance is reached [3]. The warm-up phase of a JVM has to perform a fair amount of work during runtime. This amounts to identifying and loading the necessary classes, identification of "hot" code (addressed further in Section 1.3), code verification and alike [39]. These mechanisms lead often to a "cold" start for microservices, which in turn increases the warm-up times and leads to higher memory footprint. The warm-up time is a crucial aspect to tackle for microservices, especially since they often can have short life cycles [39]. By reducing the time spent in the warm-up phase, the overall system can minimize delays and avoid breaking service-level agreements (SLAs).

GraalVM Native Image [39] is a common tool for development of microservices. It provides a lot of mechanisms looking to optimize both application performance (AOT-compilation) and workflow (polyglot programs). When it comes to the issue of warm-up time, the GraalVM Native Image aims to reduce high-memory footprint and slow warm-up times through smart AOT-compilation[1] [39]. Microservices built using the

[1] AOT-compilation is the compilation of a high-level language to native machine code.

GraalVM Native Image have been shown to have 50 times faster start-up, and 5 times lower memory-footprint compared to JVM-based solutions [30]. However, the issue which arises with GraalVM Native Image is that performance related to throughput and latency can become comparably worse to modern JVMs given enough execution time [36]. To close this gap, we introduce the Runtime Object Lifetime Profiler (ROLP).

ROLP [9] is an object profiler, tracking object lifetime information and propagating it to the garbage collector (GC). By identifying objects with different lifetimes, ROLP can inform the GC to allocate objects in separate memory-bound locations. This is known as **pretenuring**, which we explain in more detail in Section 2.4. This, in turn, reduces object copying and promotion, both of which have been shown to produce long application pause times and have a negative throughput impact [6].

On one hand, ROLP was first evaluated and integrated with the open-source garbage collector NG2C [8] for OpenJDK 8, and has shown to drastically reduce long-tail latencies of distributed big data applications [9]. On the other hand, the Native Image runtime provides a rudimentary garbage collector implementation with no object pretenuring. By integrating ROLP into the Native Image runtime, we show that the performance of microservices built using the GraalVM's Native Image is improved in regards to memory management decisions. A result of improving the GC memory management decisions is the increase in throughput and lower latency for applications built using ROLP.

1.1 Objective

In this thesis, we propose ROLP for improving the throughput and reducing the latency of microservice-based applications built using the tool provided under GraalVM known as the Native Image [39]. This is done through the use of ROLP [9], which propagates object lifetime information enable smart dynamic memory management decisions for the native runtime.

ROLP is integrated into the Native Image runtime known as *SubstrateVM*. Since the applications we will be working on are AOT-compiled to executables known as native images, a number of issues are investigated. These are related to the fact that we are dealing with applications compiled to executables and that the internal GC algorithm of SubstrateVM will require modifications to work with pretenured objects. This is covered in more detail below (see Section 1.3).

Our implementation of ROLP for Native Image is evaluated showcasing improvements in the application latency and the overall throughput. The evaluations from the experiments are covered in Chapter 6.

1.2 Requirements

The requirements of the new ROLP + Native Image version are as follows:

- accurate pretenuring of objects in correlation to their allocation site and lifetime;
- good throughput during application execution in comparison to a native image built without ROLP;
- low response time during application execution in comparison to a native image built without ROLP.

1.3 Difficulties

The GraalVM project provides several different tools such as the GraalVM Compiler [31], GraalVM Native Image Builder [39] and the Truffle Language Implementation Framework [27]. In this thesis we are focusing mainly on the Native Image Builder. Native images can be built using all of the JVM-based languages such as Java, Scala, Clojure and Kotlin [39]. The native images can also be built using languages supported by the Truffle Framework. Modifying ROLP to include support for Truffle-based languages is outside the scope for this thesis. However, implementing support for dynamically typed languages can be explored in future work.

The implementation of ROLP for GraalVM Native Image has some challenges that are addressed during the course of this thesis. In the initial work proposing ROLP, only objects which are allocated under JIT-compiled code are profiled. Since native images are AOT-compiled, we have adapted our solution to profile almost all object allocation which occur. The details for how we profile objects and deal with object allocation is covered in Chapter 4.

We have also extended the current native GC of SubstrateVM, since the current garbage collection algorithm does not deal with object pretenuring. This is due to the fact that the GC will need to take advantage of the lifetime information provided by ROLP. This in turn requires some modifications to the internal GC such that it can pretenure long-lived objects.

1.4 Thesis Overview

The rest of this thesis is organized as follows.

Chapter 2 briefly introduces the concept of a microservice, and how it compares to previous similar architectures. We will also provide

background on GraalVM (especially the Native Image runtime), object pretenuring and the profiling algorithm ROLP.

Chapter 3 presents the most relevant works which is related to the problem of reducing GC times and improving throughput through object pretenuring. The chapter is concluded with a condensed comparison of the different algorithms.

Chapter 4 explains the general architecture of the proposed solution of integrating ROLP with Native Image. We give a general overview of the functionality of ROLP, and the main data-structures that come along with it.

Chapter 5 provides a more in-depth look at the implementation of ROLP with SubstrateVM. We start of by giving a overview of the tools used, such as programming languages and build options. Then, we go into greater detail about how we modify the compiler and the native runtime.

Chapter 6 presents and discusses the results from the evaluations of running ROLP with Native Image.

Chapter 7 consists of the conclusion. Here we discuss if we have met the requirements we have set, any relevant future work, and finish of by summarizing what we have contributed.

Chapter 2

Background

In this chapter we start of by presenting microservices, and in what ways they diverge and relate to service-oriented architectures (SOAs). We then introduce the issue of warm-up time for JVM-based microservices and how it affects their performance. We show how the GraalVM Native Image can be used to greatly mitigate issues related to warm-up time. Before introducing object pretenuring, we give a general overview of what a GC is. We present in detail the runtime of the Native Image and how it manages memory. Finally, we introduce ROLP and give a brief overview over its functionality.

2.1 Microservice-oriented architecture

Microservices are considered to be based on SOAs [19]. They are usually lightweight, both in the size and functionality they provide. Some of the aspects in which microservices differ from standard SOAs are that:

- they usually rely on lightweight protocols such as REST [29] and HTTP [16] for communication, which eases integration with web based applications;
- SOAs usually rely on enterprise software, whilst microservices are often built using lightweight and open-source technologies;
- in addition, SOAs are used as an integration solution, whilst microservices are used as bounded[1] individual applications [19].

Microservices are loosely coupled, which allows them to be combined to build complete distributed systems. Since they communicate using common protocols, the technology used for creating the services can often be chosen independently of the other services [19].

[1] Minimal amount of dependencies.

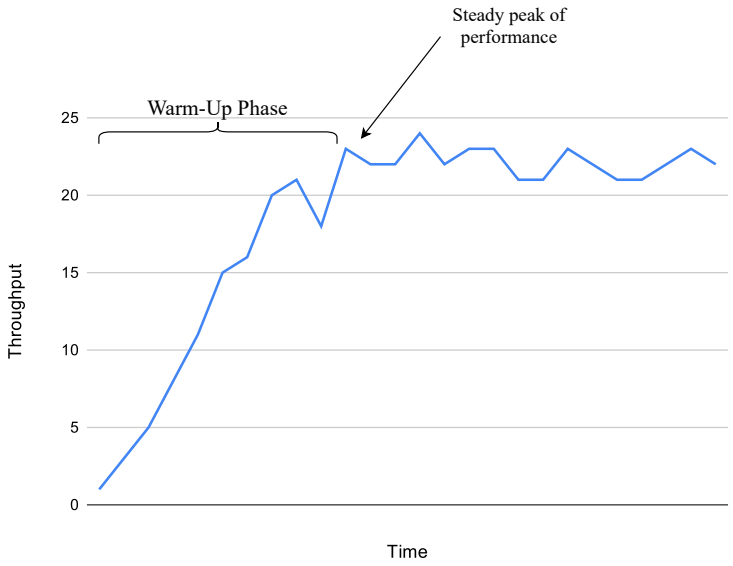


Figure 2.1: Warm-up of long-running JVM-based applications.

The design of microservice-oriented architectures provide several benefits. First, since they are lightweight and loosely coupled, working with them can be rather flexible in terms of the technology used to develop them, easing coordination for developers. In addition, the horizontal scaling allows instances of services to be scaled up by demand [18]. Server costs can also be comparably lower to a monolith solutions, given the right environment [38].

Several prominent tech companies have adopted the use of microservice-based systems. Netflix has adopted the use of microservices, by refactoring their monolithic solutions to microservice-style implementations. The same goes for Amazon [35]. There are several reasons for migrating to microservice-oriented solutions for these companies. Some of these are based on the benefits we have already mentioned: scalability, team workflow, DevOps support, etc. [33]. However, a common issue with microservices is that the combination of being lightweight and on-demand makes the cumulative time performing warm-up rather high, potentially lowering overall throughput.

2.1.1 Warm-up time for Microservices

The warm-up time of microservices is a critical aspect to tackle, especially in JVM-based services. In fact, if services were not delayed by the warm-up phase, then the whole system might be able to scale up quickly without breaking any potential SLAs. In our work, we consider the warm-up time to be the period between the startup of the VM until steady peak of performance is reached. Start-up time is the time until the first noticeable

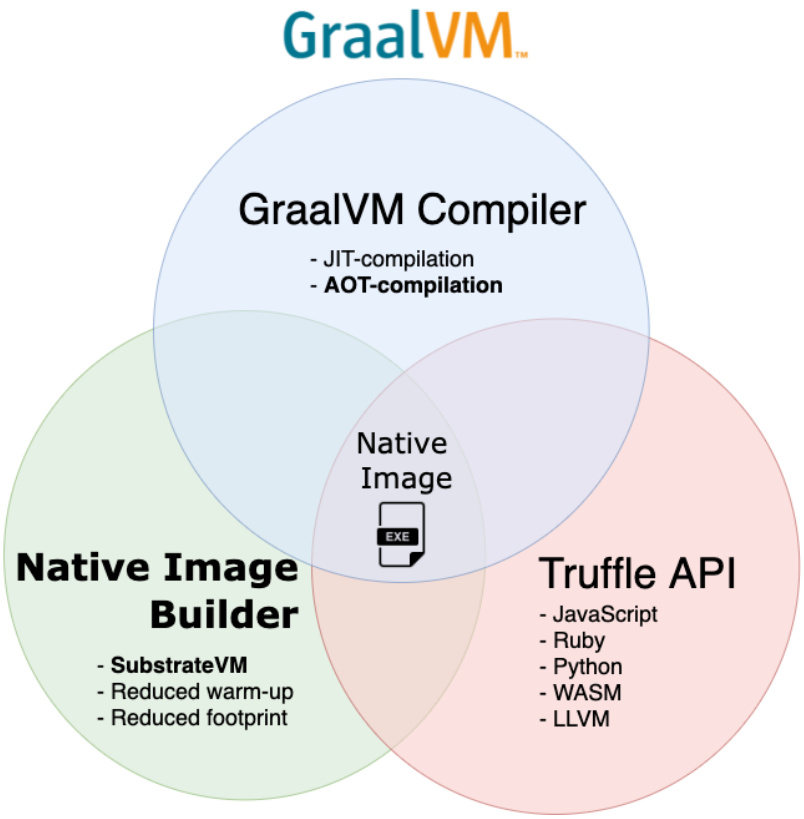


Figure 2.2: Core projects of the GraalVM ecosystem.

operation by the application is performed. In Figure 2.1 we can see how throughput over time for a JVM-based service might look. During the JVM warm-up phase, the VM has to profile code such that frequently executed code blocks are JIT-compiled. In addition, classes are to be loaded into memory when the application demands it [39]. This leads to a so called "cold" start, which can impend upon the overall performance of a system. Utilities such as GraalVM's Native Image aim to solve this.

2.2 GraalVM Native Image

GraalVM is an open-source[2] JVM and JDK maintained and under development by Oracle Labs. The GraalVM ecosystem consists of several projects in which some of the core ones can be seen in Figure 2.2. These are the *GraalVM compiler [15]*, the *Truffle Language Implementation Framework [41]* and the one which we will be focusing on during the course of this thesis, the *GraalVM Native Image [39]*.

Through the Native Image, GraalVM offers capabilities to improve the

[2]Excluding the enterprise edition.

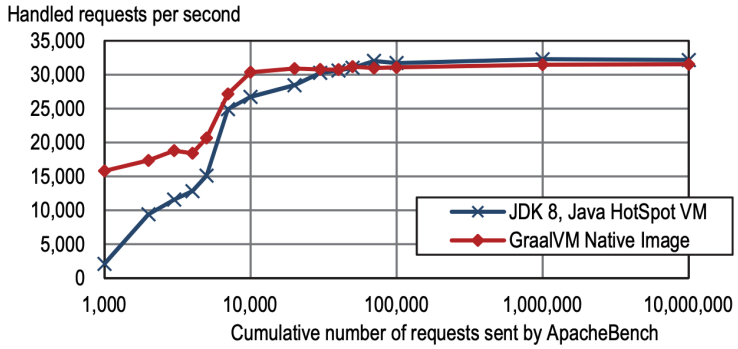


Figure 2.3: Peak performance of a REST service built on top of the Quarkus microservice framework [39].

overall performance of microservice-oriented applications, especially by targeting some of the crucial aspects for their optimal performance, which is reduction of warm-up time and memory footprint. The Native Image Builder produces a standalone executable known as a native image. This is done by GraalVM through static analysis techniques involving points-to-analysis, static code initialization, heap-snapshotting and AOT-compilation [39]. In addition, languages built on top of the Truffle Framework can also be compiled into the native image executable. This allows the applications to utilize many different libraries and tools, and at the same time ease developer coordination.

GraalVM Native Image can be already used with several different Java microservice frameworks. Some of these frameworks such as Quarkus [32], Spring [13], Micronaut [23] and Helidon [28] have already builtin support for GraalVM Native Image. The services built using these frameworks together with the Native Image provide benefits in terms of reduction of warm-up time, memory footprint and allowing developers to create service-based applications with a significant amount of tools.

2.2.1 Native Image Builder performance

Native Image offers impressive metrics when it comes to the reduction of application warm-up time and memory footprint. For some applications it has shown to have as much as a 50 time reduction in time spent performing warm-up, and 5 time reduction in memory footprint[3] [30]. In Figure 2.3 benchmark results are presented for a JSON encoder/decoder, built on top of the Quarkus framework. The results are presented in *Initialize Once, Start Fast: Application Initialization at Build Time*, a work describing GraalVM Native Image and its use cases [39]. We can see here that the native image manages to reach a steady peak of performance

[3]The amount of memory referenced and used during application execution

a fair amount of time before the corresponding HotSpot application. Throughput during warm-up is also largely better. The reason for the dip in the native image warm-up phase is because the runtime needs to claim memory for usage, which initially induces a slight overhead [39].

However, when the service is long-running, a JVM-based solution will outperform the native image given enough time. This can already be seen in Figure 2.3, in which even though the native image manages to stay fairly close to the HotSpot application, it still falls a slight bit under. To bring the performance closer to JIT-compiled applications a variety of techniques need to be employed. One of these which come along with GraalVM is profile-guided optimization (PGO) [36, 39]. PGO functions as form of dynamic analysis, in which a sample of a profiled run is analyzed, to improve the performance of subsequent compilations of the same image. For some applications, it brings the overall performance closer to a JIT-compiled version, but still can often fall short of matching the performance [36]. In addition, PGO is only enabled for GraalVM Enterprise Edition (GraalVM EE). To further improve upon the overall performance of the Native Image Builder, one can utilize **object pretenuring** (presented in Section 2.4).

2.3 Garbage Collection

Garbage collection is a dynamic memory management technique used by a variety of runtimes to automatically handle memory management decisions [20]. It was invented initially targeting *Lisp*, a dynamically typed functional language [25]. Nowadays, most runtimes based on high-level programming languages utilize some form of GC algorithm to ease development effort and avoid the variety of issues which arise from manual memory management [21].

A variety of GC algorithms exists, each of them bringing a set of advantages and disadvantages in comparison to each other. The runtime of Native Image utilizes a **generational GC** algorithm. Generational GCs were first documented in the article *Generation scavenging: A non-disruptive high performance storage reclamation algorithm* [37]. In generation based GCs the heap is separated into memory-bound spaces. Two major spaces are always included. These are the **young generation** and the **old generation**[4].

- **Young generation** is where newly allocated objects are placed into, and is often referred to as the allocation space. The size of the young generation is usually kept relatively small to the heap. The need for a

[4]These generations are further divided into **from** and **to** spaces. Covered in Section 2.5.2.

specific generation in which new objects are allocated into stem from the so-called *weak generational hypothesis*, stating that most objects tend to be short-lived, meaning most of them do not survive a single collection [37]. Thus, by allocating all objects in one region one can potentially avoid the majority of copying operations, with only a minority of objects being promoted into the old generation. This follows if most objects do behave accordingly to the *weak generational hypothesis*. When the amount of objects allocated into the young generation exceed its size, a minor collection is triggered. A minor collection promotes objects from the young generation into the old generation, leaving the unreachable objects behind for cleanup.

- The **old generation** takes up the remaining heap space, and is often referred to as the survivor space. Here are all objects which have survived at least a single garbage collection. When the old generation fills up, a major collection is triggered. These collections take substantially longer time compared to a minor collection. This is due to the fact that major collections mainly only trigger because of a full heap. During these collections, all reachable objects in the old generation persist living. In addition, reachable objects are also promoted from the young generation in the old generation (minor collections are usually a part of the major ones).

The generational GC algorithm of the Native Image runtime is covered in depth in Section 2.5.1.

2.4 Object Pretenuring

Object pretenuring is the identification of objects that survive multiple collections (and are thus considered long-lived) [5, 11]. It can be utilized in generational garbage collectors, in which objects can exist in different memory-bound spaces. Objects identified as long-lived can be directly allocated into the survivor space. Without pretenuring, such objects which tend to survive a collection have to be promoted (copied) into the survivor space.

The need for object pretenuring came after generational garbage collectors became introduced. Pretenuring was first described in *Generational stack collection and profile-driven pretenuring* [11]. The authors managed to show that by identifying long-lived objects and placing them directly in the survivor space, one can prevent copying from the allocation space. This in turn reduces latency by reducing the time spent doing collections.

There are several techniques for pretenuring which require to be adapted

for specific run-times. Tracking of object lifetime[5] can either be performed either through **online** or **offline** profiling. Which one to use depends on what one is willing to sacrifice in terms of profiling overhead and accurate object lifetime information. We present related works dealing with either techniques in Chapter 3.

2.4.1 Offline Profiling

During offline profiling, long-lived objects need to be identified during an application sample run without pretenuring enabled. There are several ways to accomplish this. One can analyze the heap after execution, or modify an application to produce lifetime statistics. After analyzing the information gained, one can now instrument the application to allocate certain objects in specific generations. The main problem which arises from offline profiling is that it might not be adaptable for applications with very dynamic control-flows. If an allocation pattern for some objects suddenly change during the run in which pretenuring is enabled, it cannot be identified. Using faulty pretenuring information could lead to memory fragmentation and scanning for already dead objects, which in turn leads to substantially higher collection times.

2.4.2 Online Profiling

Online profiling requires that the identification of long-lived objects happens during the same application run as the pretenuring will occur. A variety of techniques can be used to identify which code sections to monitor. Allocation sites to profile can be identified during the compilation of the application, since allocations are often denoted in some shape (such as AST nodes). Furthermore, one can annotate specific code blocks to monitor in which significant allocations occur. To reduce profiling overhead, one can often reduce the profiling of allocations to JIT-compiled code [9]. These techniques need to be weighted against what one is willing to sacrifice in terms of lifetime accuracy and overhead. Nonetheless, the main issue which often arises from online profiling is the overhead the profiler induces. With profiling one needs to find a balance between when to profile, and when to use the gathered statistics. The benefit here is that if some runtime metric indicates that an allocation pattern has changed (e.g. sudden increase in GC pause times) one can turn on profiling to identify new lifetime statistics.

[5]We consider lifetime to be the amount of collections an object has survived. A newly-allocated object is of age 0.

2.4.3 Pretenuring Collector NG2C

NG2C (N-Generational Garbage Collector) is the underlying GC initially utilized by ROLP [9]. It was first documented in the article *NG2C: Pretenuring Garbage Collection with Dynamic Generations for HotSpot Big Data Applications* [8]. It can be considered a pretenuring collector, as it managed to leverage the information from object profilers for pretenuring. NG2C was initially evaluated with a predecessor version of ROLP known as Object Lifetime Recorder, which functioned as an offline profiler. When integrated together with ROLP, it was shown to reduce long-tail latencies for several notable open-source distributed benchmarks.

When using NG2C, a user set number of dynamic generations can be preconfigured. This is different from the standard young and old generations, which already exist in the HotSpot runtime. By having multiple dynamic generations, objects can be more efficiently grouped. These spaces are referred to as dynamic since they can grow and decrease in size, in correspondence to the amount of objects that occupy them. Threads can be bound to a specific generation, which allows for parallel allocation. By utilizing multiple dynamic generations, objects can be allocated in their age corresponding space.

NG2C was originally developed with Object Lifetime Recorder (OLR), a profiler developed together with NG2C, which allows NG2C to allocate objects according to their lifetimes. In the article *Runtime Object Lifetime Profiler for Latency Sensitive Big Data Applications* [9], an implementation of ROLP is integrated with NG2C to provide even more accurate lifetime information. This is due to the fact that ROLP, in distinction to OLR, is an online profiler. If no profiler is used, the developer has to annotate object allocations with the predicted lifetimes.

We mention NG2C since the modifications we make to the Native Image GC can be considered a simplified version of the pretenuring collector, since we only work with two generations, the young and the old one.

2.5 SubstrateVM

SubstrateVM is the runtime behind the GraalVM's Native Image. We now present the dynamic memory management techniques it uses and the heap model of the runtime.

2.5.1 Serial GC

The default GC for both GraalVM commercial edition and EE is Serial GC. It is a simple generational, non-concurrent, non-parallel, stop-the-world[6] garbage collector optimized for small heap sizes [26]. Since it works on a single-thread, it is not preferred for parallel applications. Native Image provides a fair amount of runtime/build options which allows the developer to tweak the GC behaviour according to their needs.

As mentioned in Section 2.3, Serial GC functions as a generational GC. It contains two major generations, the young and the old one. By default, no objects are placed directly into the old generation. Serial GC does not implement any form of object pretenuring. All objects are directly allocated into the young generation, and then potentially promoted to the old generation. Objects within the old generation which subsequently survive any further collections remain in the old generation.

Before discussing the GC algorithm and heap model of Serial GC any further, we must briefly mention that the version of GraalVM we are extending (20.3.0) implements a second garbage collection algorithm known as G1 GC [14] which can be enabled for the Native Image on certain platforms. We do not focus on G1 in this thesis due to the fact that it is only enabled for the enterprise edition of GraalVM. In addition, the garbage collector NG2C is an extension of G1, and has already been evaluated with ROLP for the Java HotSpot VM [9].

2.5.2 Cheney's algorithm

The collection algorithm used by SerialGC is an adapted implementation of the *Cheney's algorithm* [10]. In this algorithm, the heap is divided into two spaces, known as the **from-space** and the **to-space**. The from-space is where objects are allocated into. Since this is the allocation space, objects can be allocated in such a way in which memory might become fragmented (due the varying size of objects). Thus, when a GC takes place, reachable objects are copied into the to-space making sure that there will be no memory fragments. At the end of the collection, the to-space becomes the new from-space.

This object copying extends each original object in the from-space with a so called **forwarding pointer**. The forwarding pointer informs the GC if the object has a copy of itself in a chunk in the to-space. These objects are in addition marked with a forwarding bit, indicating that they do not need to be scanned through again (when looked for references to other objects). The benefits of the Cheney's algorithm is that it minimizes memory fragmentation, and only needs to keep track of the survivor

[6] A GC which halts program execution until the collection is finished.

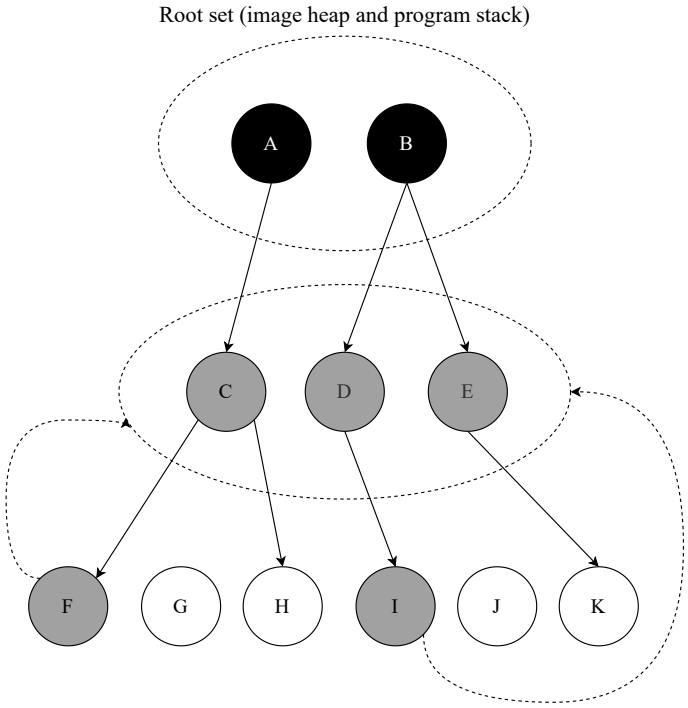


Figure 2.4: Tri-mark coloring for objects in Serial GC.

objects. Dead objects are left behind in the from-space, such that we remain with only reachable objects in the to-space.

In SubstrateVM, since we have two generations, these need be divided into from and to-spaces. The young generation contains a single space known as the *eden space*, which functions as a from-space.[7] The old generation is equally divided into *old from-space* and *old to-space*. During minor collections, reachable objects are promoted from the eden space to the old to-space. During major collections, reachable objects both from the eden space and the old from-space are promoted into the old to-space. At the end of a collection the old to-space becomes the old from-space.

2.5.3 Reference scanning

In addition to utilizing the cheney-scan algorithm, Serial GC uses the *tri-marking* [20] color abstraction to denote if objects are reachable or not. Objects are denoted as either black, white or grey. The color correspond to the liveness of the objects.

- All objects are initially considered as **white** (besides the object references on the program stack and the image heap, which are

[7]It is further possible to divide the eden space into several buffer survivor spaces. This is not considered in this thesis.

always initially considered as grey). Objects at the end of the collection which are white are considered unreachable (dead).

- **Black** objects are those that are considered live at the end of the collection. These are objects within the image heap, stack and the to-space at the end of the collection.
- **Grey** objects are those which are reachable (live) but have not been scanned for references. Grey objects can become black if all of their references are also scanned. The scanning of grey objects is where a significant amount of time is spent for a collection in Serial GC (up to 99% of total collection time in some cases, since this is where copying takes place).

Collections always begin by marking objects which are guaranteed to be alive. These are objects which either exist on the image heap or the program stack. These are the objects we have denoted in the top set in Figure 2.4. References that have been found through the top set are considered reachable and thus denoted as gray. This is the set below the root set. These gray objects are then promoted into a to-space, and their reachable references will be added to the workset of objects to be scanned. At the end of the scanning, if any objects remain in the from-space (G and J in in Figure 2.4), they are considered dead (white). The space which these dead objects occupy is consumed and either claimed by the operating system or reused by Serial GC as future chunks of memory.

2.5.4 Heap Chunks and TLABs

The spaces which are used in SubstrateVM's heap model consist of heap chunks. These are (by default) of 1024KB, and contain a continuous segment of virtual memory dedicated to objects. The spaces contain a collection of heap chunks formed as a linked list, with the tail chunk being the latest one. These chunks are categorized as either **aligned** or **unaligned**.

Aligned heap chunks can hold multiple objects continuously. The objects can be mapped to their respective parent heap chunk. For an aligned heap chunk, an object promotion happens with the object being moved from the parent heap chunk to a target heap chunk in a old to-space.

Unaligned heap chunks can hold one large object (in Serial GC only large arrays), and thus all unaligned heap chunks are allocated through the slow-path (how allocation paths work for both type of chunks is described in the following paragraph). Since they only contain one object, the whole heap chunk is moved during promotion.

A GC is triggered based on the amount of heap chunks we have allocated

since last GC (which is in relation to how many objects have been allocated since a GC). New heap chunks can be generated during on object allocation. An object allocation can go through two paths:

- The **fast-path**, in which an object is allocated directly into a heap chunk using the offset value in the chunk known as the **top**. The top of the heap chunk is the current memory address in which the object will be allocated into. If `sizeof(obj) + top` is not beyond the offset value which is the end of the chunk, we allocate the object. This functions as a form of bump-pointer allocation, which is relatively inexpensive and tries to waste minimal memory space (and therefore perceived as the fast-path).
- If the object allocation would cause the heap chunk's new top to be higher than the end of it, we go through the **slow-path**. This means that a heap chunk does not have enough space for the object, and we either need to allocate a new one or retrieve one from a free-list of unused chunks. After a new heap chunk is generated, we perform bump-pointer allocation on it for the new object. After enough slow-path allocations a collection will trigger. Right before the collection has begun scavenging for live objects, the chunks are assigned (flushed) to a space depending on their parent Thread-Local Allocation Buffer (TLAB).

TLAB covers a region of heap chunks within the allocation space in which allocations can occur. When a new chunk is assigned to a TLAB (due to the slow-path), it will become the active allocation chunk for the specific TLAB. The TLAB keeps track of all of the chunks assigned to it as a linked list until a collection is triggered. When a collection is triggered, the chunks are flushed from the TLAB and assigned to a space. Serial GC contains a single TLAB which points to the eden space.

2.5.5 Card and First Object Table

Serial GC has two types of collections, **incremental** (minor) and **complete** (major). Incremental collections are responsible for collecting and promoting objects in the young generation, whilst complete collections perform the equivalent for both the young and old generation. Since an incremental collection takes care of objects that have references which can be followed into the young generation, the scanning algorithm has to take regard for objects in the old generation which point to objects in the young one. A naive approach would be to scan through the live roots in the old generation, and by following those references promote young generation objects into the old generation. Serial GC handles this more efficiently through data-structures known as **Card Remembered Set Table** and **First Object Table** (FOT).

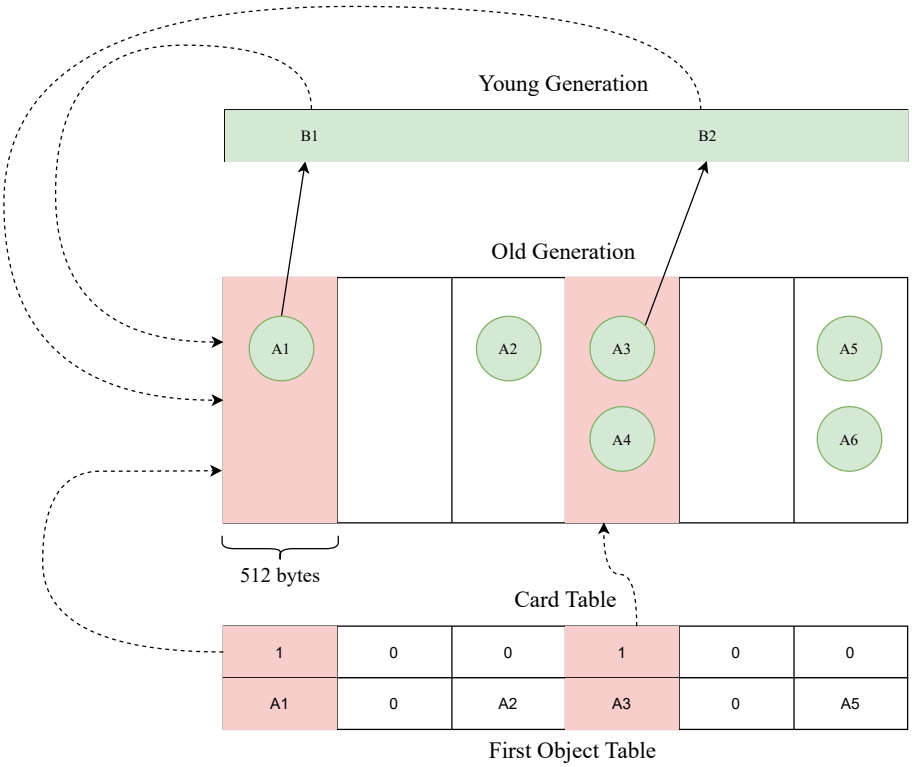


Figure 2.5: References in the young generation from the old generation being tracked through card tables.

Instead of scanning the entire old generation for references to the young generation during incremental collections, the Card Table summarizes which memory regions in the old generation can contain references to the young generation. Serial GC does this by setting a **dirty** byte in a index in the card table. We can see this in the bottom of Figure 2.5, in which two dirty bits are set for the card table. This in turn informs an incremental collection to scan the old generation memory regions of objects A1 and A3. During these scans, references to objects B1 and B2 will now be found and promoted to the old generation. The regions containing A2, A5 and A6 are not necessary to scan since the card table indicates they do not hold any references to the young generation. In Serial GC the regions to scan for consist of a default size of 512 bytes each.

The FOT simply allows the header of the first object in the dirty region to be located instantly instead of being scanned through from the beginning. The header of an object consists of a **remembered set bit** if it could hold a reference to an object in the young generation. In Figure 2.5, the FOT would point to objects A1, A2, A3 and A5 given they are the first objects in their region. The card table and FOT are embedded into each heap

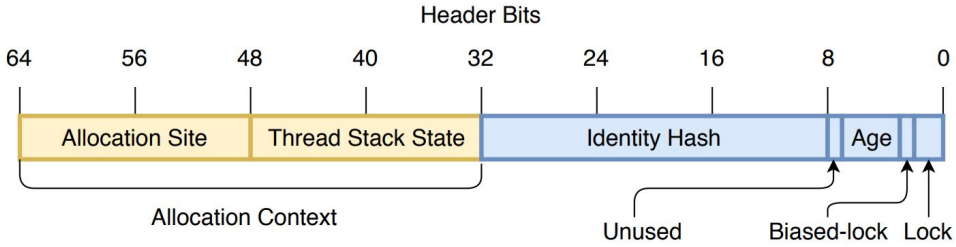


Figure 2.6: Object header in HotSpot with ROLP [9].

chunk. Unaligned chunks contain only the card table.[8]

Mentioning these mechanisms is relevant since our implementation of ROLP will be moving objects directly to the old generation. This requires us to modify object headers and declaring the FOT of each new object. The implementation of how we take care of these kind of objects is covered in Section 5.4.2.

2.6 Runtime Object Lifetime Profiler

ROLP was initially presented in the article *Runtime Object Lifetime Profiler for Latency Sensitive Big Data Applications* [9]. The goal was to improve GC pause times through online object lifetime profiling, which then is allows objects to be allocated more efficiently. The garbage collector leverages the lifetime statistics from the profiler to decide if objects should be allocated in a different allocation space.

It was originally developed for OpenJDK 8, Java HotSpot VM, and combined with the open-source collector NG2C [8], which leveraged the profiling information to allocate objects in one of the preconfigured allocation spaces. The results in the article show large reductions in long-tail latencies in several open-source benchmarks, with minimal throughput and memory overhead [9].

ROLP introduces several techniques for object profiling. The profiling is performed online, such that it can adapt to the execution of the application. To uniquely identify and group together objects, it uses their corresponding allocation site. The allocation site is an identifier for where an object allocation occurs in the application code. It is translated into a 2-byte number embedded in to each profiled object's header.

Furthermore, to be able to distinguish objects which have the same allocation site but different allocation paths, a 2-byte number denoting the

[8]Due to only containing one object which is always aligned after the card table field.

thread stack state (TSS) is used. It is embedded together with the allocation site in the object header. The object header for the HotSpot version of ROLP can be seen in Figure 2.6. The allocation site ID and TSS are both taking up unused space in the object header, thus not affecting the size of the profiled objects. The age field which we seen within the header is also updated by the collector. For keeping track of the lifetime statistics, an *Object Lifetime Distribution Table* (OLD table) is utilized.

2.7 Summary

In this chapter, we have presented why GraalVM's Native Image is utilized in the development of microservice-oriented architectures. Furthermore, we introduced how object pretenuring can be utilized to reduce the amount of work done by generational GC algorithms. We gave an in-depth look on how the Native Image GC algorithm works, together with the heap model. Finally, we gave an overview of ROLP's functionality. In the next chapter, we will discuss related work which been done in regards to object pretenuring.

Chapter 3

Related Work

In this chapter we outline similar work which has been done in optimization of microservices and other distributed systems, particularly through object lifetime profiling and pretenuring. We first introduce the first paper which presented work related to object pretenuring for generational garbage collectors. We then present POLM2 [8], the offline predecessor of ROLP. Next, we discuss related work done in terms of object pretenuring, and what shortcomings they have compared to ROLP. A table providing condensed comparisons for the different works is presented at the end of the chapter.

3.1 Generational stack collection and profile-driven pretenuring

Object pretenuring was initially introduced in the paper *Generational stack collection and profile-driven pretenuring* [11]. This paper introduces mainly two techniques. *Generational stack collection*, which is mainly relevant for functional languages, and thus not discussed in this work. For generational garbage collection it introduces object pretenuring. In much the same way as ROLP, objects are profiled through an allocation site. Objects which survive enough collections into a survivor space become targets for pretenuring. This means that these object are directly allocated into the survivor space.

Since this is the initial work introducing object pretenuring, several limitations are present. First of all, the solution is offline. The heap profile is scanned after running the application. Applications are also modified to produce lifetime statistics for each object. Then, the gathered statistics are used to determine which objects should be pretenured. The programmer has also to declare which sites to profile, which can be cumbersome for large projects. Objects allocated through particular allocations sites

are grouped to have the same lifetimes, with no consideration for the path taken to allocate them. In addition, since the solution works offline, applications with highly variable control-flows can lead to faulty pretenuring of objects.

3.2 POLM2

In the article *POLM2: Automatic Profiling for Object Lifetime-Aware Memory Management for HotSpot Big Data Applications* [7]. an offline profiler referred to as POLM2 is introduced. It is integrated with the pretenuring collector NG2C [8] for OpenJDK 8, HotSpot JVM.

POLM2 functions through four main components. These are referred to as *Recorder*, *Dumper*, *Analyzer* and *Instrumenter*. The components function through different phases which it defines as *profiling* and *production*. A short description of the phases with their corresponding components:

- *Profiling* occurs during the analysis of the sampled run and consists of the components;
    - ▷ *Recorder* which monitors object allocations, and uniquely identifies the allocation sites.
    - ▷ *Dumper* which receives a call from the *Recorder* to produce a heap snapshot of the JVM-heap.
    - ▷ *Analyzer*, which together with object allocation information and corresponding allocation sites from the *Recorder*, in addition to heap snapshot from the *Dumper*, can then produce lifetime statistics for each of the allocation sites.
- After *profiling*, we have the *production* phase.
    - ▷ It consists of a single component, *Instrumentation*. Here the the bytecode of the subsequent compilation of the application is rewritten such that objects are pretenured according to the lifetime statistics gathered from the profiling phase.

POLM2 suffers from the same issue we cover for the other offline-based profilers, which is low adaptability to dynamic control-flows. Nonetheless, ROLP can be considered the online successor of POLM2 in a lot of aspects. The combination of using both the call stack trace and an allocation site to distinguish object allocations is an example of something ROLP has imitated from POLM2 with an online version.

3.3 Learning-Based Pretenuring

In the article *Decrypting the Java Gene Pool* [24] the authors introduce a pretenuring technique based on using previously acquired lifetime statistics[1] combined with identification of certain **micro-patterns**. The article defines micro-patterns as "a non-trivial, formal condition on the attributes, types, name and body of a class and its components, which is mechanically recognisable, purposeful, prevalent and simple" [24]. The authors aim to show that there is a high correlation between coding patterns and the lifetime of objects, and by the using gathered lifetime statistics one can reduce collection times drastically. The pretenuring techniques are implemented for Jikes RVM, a research based VM maintained as an open-source project [1].

The main limitation originating from the work described is that the analysed micro-patterns and lifetime statistics have to fit any application. Thus, if the application in which one wants to enable pretenuring is vastly different in coding style from the analyzed ones, the pretenuring information might be invalid and cause inefficient tenuring of objects. A critical example of this would be wrongly denoting an allocation site as immortal.[2] In addition, coding style can vary highly between different programming languages. This means that the knowledge bank has to be updated to fit an unknown number of programming languages. A profiler such as ROLP needs only to hook up to code responsible for object allocation and promotion, with no knowledge of the programming language/pattern used (especially for the Native Image, since it supports languages built with Truffle).

3.4 Allocation Mementos

Allocation Mementos [12] is a work targeting improvement in dynamic memory management using several online techniques. It is developed by Google and implemented for the JavaScript engine V8 [2].

Allocation Mementos is one of the works that is fairly comparable to ROLP. One of the techniques used is online object allocation site profiling to identify long-lived objects. This is done through small objects which are referred to as *mementos*. These objects enable allocation site profiling in much the same way as ROLP, by tracking newly allocated objects and survivor objects. This is done by connecting the memento objects to a profiler which contains information about how many times objects have been allocated and promoted through a particular allocation site. Based

[1] The lifetime information is acquired from the DeCapo benchmark suite [4].

[2] Objects in allocation sites deemed immortal are allocated in a region never collected.

	Latency	Overhead	Profiling	Runtime
POLM2 [7]	Up to 80% reduction in GC pause times for several workloads.	No memory and throughput overhead.	Offline	OpenJDK 8 HotSpot JVM.
PDP[3] [11]	Reduction in GC times by 12-50% for certain benchmarks.	NA	Offline	TIL [34]
Mementos [12]	Targets latency for the V8 engine.	Arises from the injected mementos objects. (Slowdown of 3% for one the benchmarks.)	Online	V8 [2]
LBP[4] [24]	6-77% reduction in pause times for spec jvm98.	NA	Offline	Jikes RVM [1]

Table 3.1: Summary table for the relevant works.

on the information gathered in the allocation site profiler, objects can be directly allocated in the old generation.

The work further introduces several other techniques to optimize memory management. One of them being array **pretransitioning**. Array transitioning is an optimization technique which changes array element representation. Changing the representation is done to optimize later operations performed on a specific array (considering both time and space). Since transitioning is an expensive operation, pretransitioning tries to optimize this by predicting the optimal array element representation type. The last of the major optimization techniques this work introduces is **presizing**. This consists of initializing an array to a size which prevents further element copying and array resizing.

[3]We refer to the pretenuring done in *Generational stack collection and profile-driven pretenuring* [11] as PDP.

[4]We refer to the work in *Descrypting the Java Gene Pool* [24] as LBP.

In comparison to the original ROLP, Memento does have some limitations. First, mementos do not track the call path of objects, which in ROLP is done through allocation context. This can lead to insufficient profiling information. Furthermore, an overhead cost is introduced, especially in memory footprint, by the *mementos* objects. ROLP does not require any further object allocation for object profiling. It simply installs the unique allocation site identifier in an unused field in the object header.

3.5 Summary

We present a condensed comparison of the different related works mentioned above in Table 3.1. The table displays the latency reduction for target environment, profiling overhead, if the profiler is offline or online and the target runtime in which the mechanisms are implemented. In the next chapter, we propose several techniques and modifications to enable object pretenuring for the Native Image through the usage of ROLP.

Part II

The Architecture and Implementation.

Chapter 4

The ROLP-SubstrateVM Architecture

In this chapter we provide a general overview of the architecture of ROLP on top of SubstrateVM and their role within Native Image. We especially focus on the required modifications to the builtin garbage collector Serial GC and the object lifetime profiling techniques we propose.

4.1 Serial GC with Object Pretenuring

There are several modifications which are needed for the native collector Serial GC to enable object pretenuring. We present these in the following sections.

4.1.1 Introducing a Second TLAB

To manage object allocations directly to the old generations, we propose the usage of a second TLAB. We refer to it as *OldTLAB*, to distinguish it from the default one for the young generation, which we refer to as *YoungTLAB*. Allocations which happen through the *OldTLAB* place objects in the old generation. Only objects which ROLP identifies as long-lived are allocated through the *OldTLAB*.

OldTLAB can be seen in the bottom old generation of Figure 4.1. We can see that *OldTLAB* covers heap chunks within the old from-space. Besides distinguishing between allocations for the young and old generation, allocations which would have lead objects to be placed in unaligned chunks are ignored by *OldTLAB*. Unaligned chunks are largely uncommon, since applications do not tend to allocate a high amount of large objects. Together with the fact that these chunks only contain one object, we do not consider them important for profiling and pretenuring.

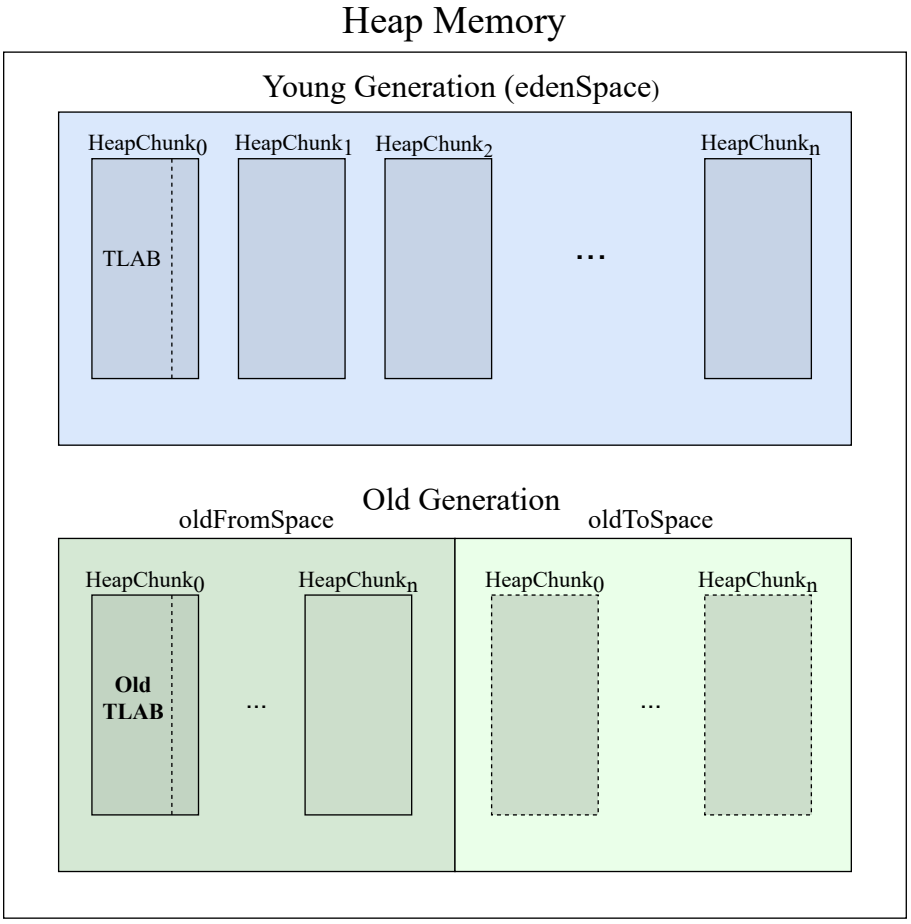


Figure 4.1: Basic SubstrateVM's heap model. **OldTLAB** in the old generation is an addition of our GC modification.

The necessary modifications for *OldTLAB* are not a significant overhaul, as it mostly entails just changing the path of some heap chunks allocations from the eden space to the old from-space. How we determine if some chunks should be allocated in the old from-space through *OldTLAB* is determined by looking up object lifetime distribution in the OLD table (see Section 4.2.5).

4.1.2 Modifying the Collection Policy

The (default) policy set by Serial GC is *ByTime*. This policy tries to balance the time spent between doing incremental and complete collections. This is mainly based on two conditions. If the collective amount of time spent doing young collections is above a threshold, the next collection will be complete. In addition, if the amount of space occupied by objects after the previous collection is above a certain threshold, a complete collection is also triggered. If the conditions above are not met, only an incremental

collection is performed.

A collection is triggered only during slow-path allocation (see Section 2.5.4). The size of the newly allocated heap chunk is counted towards checking if the total size of all heap chunks in the young generation exceeds its maximum size. If it does, a collection is triggered. As mentioned above, this collection will be only incremental if none of conditions of the `ByTime` policy are met for a complete one. A complete collection would entail also an incremental one.

Given our proposal of ROLP, an issue arises which is that objects which are directly allocated in the old generation should not count as heap chunks for the young generation. However, we require to keep track of them for a collection for the old generation. For this, we propose the usage of a second counter of heap chunks allocated through *OldTLAB*, which is used to trigger a collection based on heap chunks allocated through the slow-path for the old generation. The condition to trigger a collection based on these new heap chunks is that their total size exceeds a certain percentage of total heap space. Collections triggered through this path cause complete collections. The implementation of this mechanism is covered in detail in Section 5.5.2.

4.2 Integrating ROLP

Designing ROLP for the Native Image runtime SubstrateVM implies providing the information regarding object lifetimes to Serial GC. Objects showing a pattern for surviving at least 1 collection are considered **old** and allocated directly in the old generation. Objects that do not tend to live passed a single GC are categorized as **young** and allocated in the young generation. Statistics regarding if an object is old or young is gathered by tracking objects promoted and allocated through corresponding allocation site. We keep track of the lifetime statistics in the the OLD table (see Section 4.2.5).

The complexity of our version of ROLP is considerably lower than the HotSpot version, since we are only working with two allocation spaces, compared to the configurable amount of ROLP + NG2C. Serial GC is also non-concurrent and thus all of the GC work happens through a single thread. In the following sections we present the specific algorithms and data-structures required to enable object pretenuring in Serial GC through the usage of ROLP.

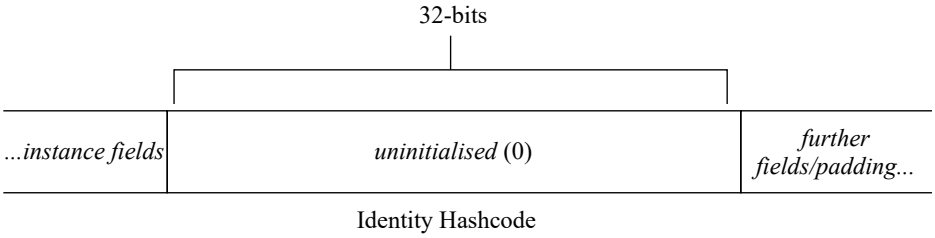


Figure 4.2: The bits in the object header which we are going to occupy. By default it contains the (uninitialized) identity hashcode.

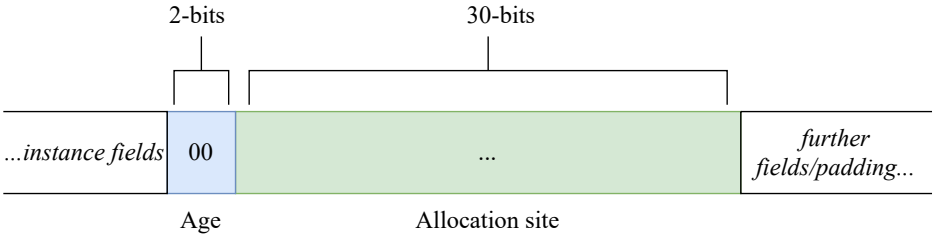


Figure 4.3: The bits we are occupying within the object layout for the allocation site identification and age.

4.2.1 Profiling of Objects

We propose profiling of objects through their corresponding allocation sites. Allocation sites functions as an unique identifier for the code position of where objects are allocated. Objects allocated can then be referenced through their allocation site identifier. We calculate the allocation site identifier for each object allocation site by simply incrementing a counter for each new site. The counter is then embedded as a 30-bit number into the object's identity hashcode field.

In fact, each object´s header has 32-bits (initially reserved for the object identity hashcode) in which, the 2 left-most bits, are used by us for tracking the age of an object, and therefore initialized to 0. As already mentioned, the space we occupy within an object´s header is initially reserved for the object identity hashcode. This can be seen in Figure 4.2. The hashcode of an object is generated on request; therefore, the space reserved for it can be safely used for the allocation site identifier until it is required. The usage of the hashcode field can be seen in Figure 4.3. How we differentiate between valid allocation sites and hashcodes is described in Section 5.5.1.

4.2.2 Rewriting Compiler Intermediate Representation

```
Function SetupObjectAllocationSites(graph):
    foreach NewInstanceNode n ∈ graph do
        /* finding the hashcode offset within object layout */
        hashCodeOffset = n.getHashCodeOffset();
        address = OffsetAddressNode(n, hashCodeOffset);
        /* inserting allocation site in object              */
        allocationSite = createAllocationSiteForNode();
        writeNode = WriteNode(address, allocationSite);
        /* including the changes to the IR graph            */
        graph.add(address, writeNode)
    end
End Function
```

Algorithm 1: Instrumenting allocation nodes to include allocation site.

The compiler is instrumented to embed the allocation site ID into the objects. This requires the rewriting of the intermediate representation (IR) of the compiler. The compilation of native images works through compilation phases which modify the Abstract Syntax Tree (AST). We propose the addition of a custom compilation phase to modify compilation nodes representing allocations, such that their corresponding objects contain the unique allocation site identifier.

The algorithm modifying the compiler is illustrated in Algorithm 1. The method shown must run after all of allocation nodes have been inserted into the compilation graph. We refer to each one of these allocation nodes as a `NewInstanceNode`. We setup the allocation site for each of the nodes in the foreach-loop of `SetupObjectAllocationSites`. Since we are going to occupy the identity hashcode field in the object header, we first need to fetch the hashcode offset of the object. This is transformed into a node representing the hashcode offset address for the given object allocation node. A unique allocation site identifier is generated which is inserted into a node together with the offset address for the identity hashcode. This is done such that the nodes we made are a part of the compilation IR graph; we append them to the graph after all necessary changes have been made. During runtime, all object allocations will now have an allocation site value written into them in the identity hash code field.

4.2.3 Profiling Phase

To gather lifetime information about objects, we restrict a pre-defined number of epochs[1] from the beginning of the program execution for profiling. During this phase, only information regarding object lifetimes is

[1] Epoch corresponds to a GC cycle.

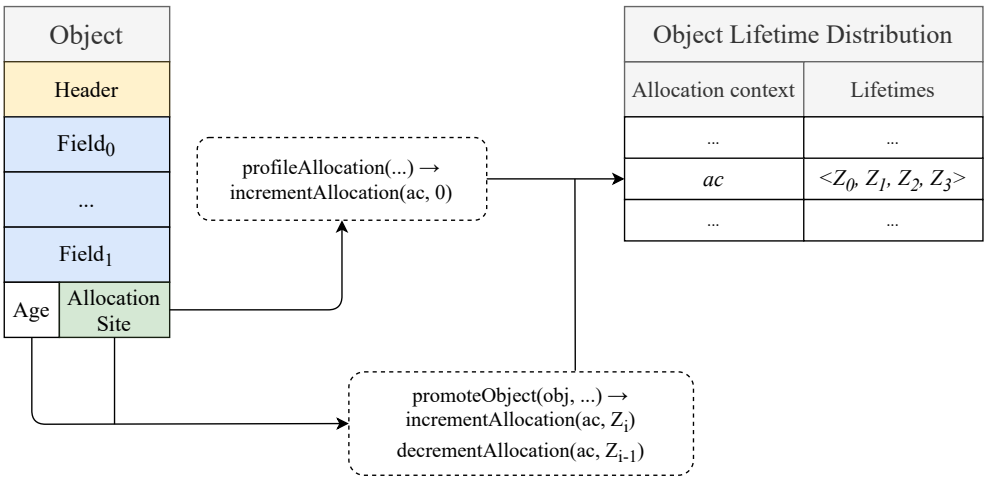


Figure 4.4: Depiction of the events triggering an update in the OLD table. `profileAllocation` is called upon object allocation. `promoteObject` is called upon object promotion.

gathered, without pretenuring objects. How this information is gathered is covered in the subsequent sections.

After the GC has passed the final profiling epoch, profiling is turned off to enable pretenuring of objects. Profiling is also turned of to restrict costs related to it, since it does induce a significant overhead.

4.2.4 Object Allocation

During object allocation, it is decided if we should allocate the object in a heap chunk designated for the young or old generation. This is done after the profiling phase, which is determined by a user set epoch. After the profiling phase, we look up the distribution of the given allocation site attached to the object. If the number of promoted objects for the allocation site outnumber the number of newly allocated objects, the object is allocated in active *OldTLAB* chunk. Otherwise, the object is allocated in the active chunk of the *YoungTLAB*. The reasoning for waiting until above a certain GC epoch, is because we wait until we have a fair distribution of the objects in the OLD Table (described in Section 4.2.5). Looking up object target generation too early could lead to objects being allocated in the wrong generation.

4.2.5 The Object Lifetime Distribution Table

To keep track of newly allocated and promoted objects for a given allocation site, a data-structure is needed. We propose a simple *Object Lifetime Distribution Table* to keep track of object lifetime statistics. Each entry in the table corresponds to a lifetime distribution for an allocation

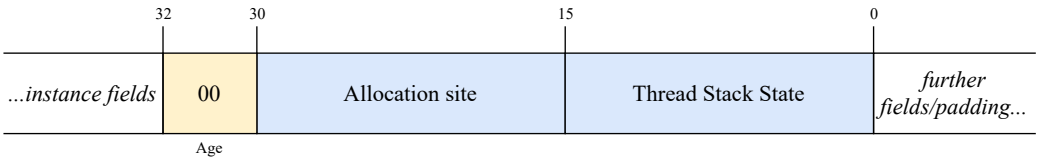


Figure 4.5: Allocation context with Thread Stack State.

site. The lifetime of objects is calculated in terms of how many GC cycles they have survived. This can be calculated up to a maximum of 3, since 2 bits in the modified object header are reserved for this, as shown in Figure 4.3. The size of the table is pre-initialized to a static size.

The lifetime distributions within the table are primarily updated during two events as described now.

- **Object allocation**, in which a new object is generated. In this case we simply increment the amount of objects of age 0 for a given allocation site. This is done for each allocated object during profiling.

- **Object promotion**. This is what allows to determine if most objects are surviving into the old generation. An object is promoted if it survives a collection from a from-space into a to-space. If an object is promoted, we increment the amount of objects with age i for the corresponding allocation site. We then decrement the amount of objects of age $i - 1$. The age bits within the object layout are also modified to match the new age of the object. This is done for each promoted object in aligned chunks during profiling.

There are further minor events which could trigger an update in the OLD table. These are are mainly:

- **Generating hashcode**. When an object requires a hashcode, the allocation site value has to be overwritten. This in turn decrements the amount of objects in the table with a given allocation site value, since the object is not longer being tracked.

- **Tracking deallocated objects**. Since all of the heap chunks in the from spaces are consumed[2] at the end of a collection, we can run through them and find each object which has not been copied into the survivor space. For each non-forwarded object we decrement the amount the corresponding objects in the OLD table. This can make the lifetime information substantially more accurate.

4.2.6 Tracking Thread Stack State

To further increase the accuracy of the pretenuring statistics, we propose the usage of Thread Stack State (TSS). TSS is used to further distinguish allocations which happen through different method call paths. This is done by counting the number of method calls before the allocation of an object. The number of calls is then embedded together with the allocation site identifier in the object header. How the object identity hashcode field in SubstrateVM would look like with TSS can be seen in Figure 4.5. 30 bits are reserved for both the allocaiton site and TSS in the identity hashcode field, both occupying 15 bits each. The age occupies the two left-most bits.

4.2.7 Dynamic Profiling

Profiling from ROLP does induce a significant overhead. This is due to the fact that we have to do a memory read of the object header, and potentially a write if the object age is incremented, for each object which survives a collection. The problem of turning off the profiling is that if object lifetimes drastically changes, the previous pretenuring statistics become faulty.

This could be the case in applications which might deal with varying loads of incoming data. There are several ways to manage this. We propose the usage of dynamic profiling. Detecting a possible incorrect pretenuring pattern due to a change in object behaviour could be done by checking if there is an increase in the time spent doing collections. To determine that the collection times have increased, a technique could be to compare the average time spent doing collections after profiling has been turned of for a set amount, and then see if there is a set of collections after that which spend more time by average. If this is the case, it means most likely some objects are being pretenured incorrectly. Then one can turn on profiling again for a set amount of epochs to capture new lifetime information.

4.3 Discussion

We now present some of the divergences ROLP with Serial GC takes compared to ROLP with NG2C. As we can see in Table 4.1, some of the proposed features we see in Chapter 4 have not been pursued during implementation. These are mainly dynamic profiling and tracking of TSS. These features are not necessary for object pretenuring, but would increase the reliability of the lifetime statistics gathered and relieve developer effort.

The reason for not pursuing TSS is simply due to the complexity of the feature, which is outside the scope of this thesis. Identifying the set

[2]Released to memory or added to a list of unused chunks.

	ROLP with Serial GC	ROLP with NG2C
Runtime	GraalVM Native Image	Java HotSpot
Target Applications	Microservices	Big Data Appplications
Tracking with TSS	No	Yes
Dynamic Profiling	No	Yes
Fully accurate lifetime information	Yes (with tracking of deallocated objects)	No
Code Profiled	All allocations	All JIT-compiled allocations
Generations	Old + Young	Old + Young + 14 generations

Table 4.1: Comparison of the different ROLP versions.

of functions to track and how long to track them for is not trivial, and requires heavy experimentation.

Dynamic profiling requires mainly the tracking of a set of collection times and comparing the averages against each other, and some minor modifications to ROLP. We have not pursued this mainly because we believe that the evaluations would have not been heavily impacted by it. We discuss these features further under future work.

A feature which is available in our version of ROLP, which was not trivial for the NG2C version, is the tracking of deallocated objects for highly accurate lifetime information. We discuss the implementation details of this feature and its impacts on the profiling in section Section 5.4.5.

4.4 Summary

In this chapter we have gone through the necessary modifications required to enable object pretenuring for Serial GC. In addition, we presented how object lifetime profiling can be implemented through the usage of ROLP. In the next chapter, we present the actual implementation of the proposed modifications in detail.

Chapter 5

Implementation of ROLP with SubstrateVM

In this chapter we provide an in depth implementation description of ROLP for Serial GC. First we mention which development tools we use, such as programming languages, build tools, runtime/build options and operating system. Then, we describe how the compiler is extended to enable allocation site tracking and how the resulting compiler graph changes. We discuss the implementation details of the events triggering updates in the OLD table and how we avoid high profiling costs. Modifications to the heap model and collection policy are also presented.

5.1 Development Tools and Project Repository

We start by providing an overview of the programming languages and tools used for the implementation. We also describe the modified files, and where the implementation can be found.

5.1.1 Programming Languages

The language we use to extend SubstrateVM with ROLP is Java 8. This falls as a necessary choice, since the majority of the VM is written in Java 8. For read/write to memory and lower-level access, SubstrateVM provides the necessary wrapper methods which either use the internal class `Unsafe` or bindings to the `C/C++` languages.

The applications which we run as native images are written as pure Java 8 programs. We avoid the use of other JVM-based languages or usage of languages written on-top of the Truffle API.

All of the development is done under the operating system **macOS**

Catalina version 10.15.7. The implementation should work for all operating systems supporting GraalVM 20.3.0.

5.1.2 Build Tools and Options

To build and generate the sources for the GraalVM projects, we use the tool provided under the GraalVM project known as **mx** [17]. Through the use of `mx build` we build the projects from scratch, and generate the necessary sources to run the different GraalVM projects. To generate projects files and enable formatting for IDEs, the command `mx ideinit` is required. `mx clean` is used to remove compilation artifacts.

After building the project, we can compile applications into the native image executables. This is done by first compiling them into .class files containing java-bytecode. This is done through a regular JVM (done by running `javac`), and then passing along the main class file to the `native-image` script within the GraalVM repository.[1] Running the script should generate a native image with the runtime of SubstrateVM.

The native image compilation options which have been necessary under the development of ROLP were:

- `-H:+RolpGC` to enable ROLP.
- `-H:FinalEpoch` to specify at which epoch to stop profiling with ROLP. Default value of 16.
- `-H:+AllocationProfiling`. This option is necessary to run ROLP, since it enables us to pass along the allocation site of each `NewInstanceNode` and `NewArrayNode` to an allocation profiler incorporated for each allocation site. Without modifications, this option tracks the amount of objects that have been allocated through a specific allocation site.
- `-H:+SurvivalRate` to print the percentage of objects which survive a collection. This was added such that the we can confirm objects are being allocated according to their distribution in the OLD table.
- `-R:+VerboseGC` to print information about each garbage collection (current epoch, heap chunks, time spent, etc.). Necessary for benchmark testing. `+R:PrintGC` for more detailed logging.
- `-H:+TraceObjectPromotion` to print information regarding an object about to be promoted. Modified to include lifetime distribution for object's corresponding allocation site.

[1] The location of the script is under `sdk/latest_graalvm_home/bin`.

- `-R:PercentHeapThreshold` to adjust when to trigger a collection according to taken heap space. Added to match the heap usage of native image without ROLP.
- `-H:InitialCollectionPolicy`. The default GC policy used by SubstrateVM is `ByTime`. As already mentioned in Section 4.1.2, it tries to balance between the time spent doing collections due to collection time and occupied heap space. We implement a slightly differing policy `ByTimeWithRolp`, which in addition takes into account the occupied size within the old generation and triggers if the heap threshold from `PercentHeapThreshold` is met.

There are several build and runtime options which we cover in more detail in Chapter 6. We cover them further on as they are only relevant to the specific benchmarks and evaluations they are enabled for.

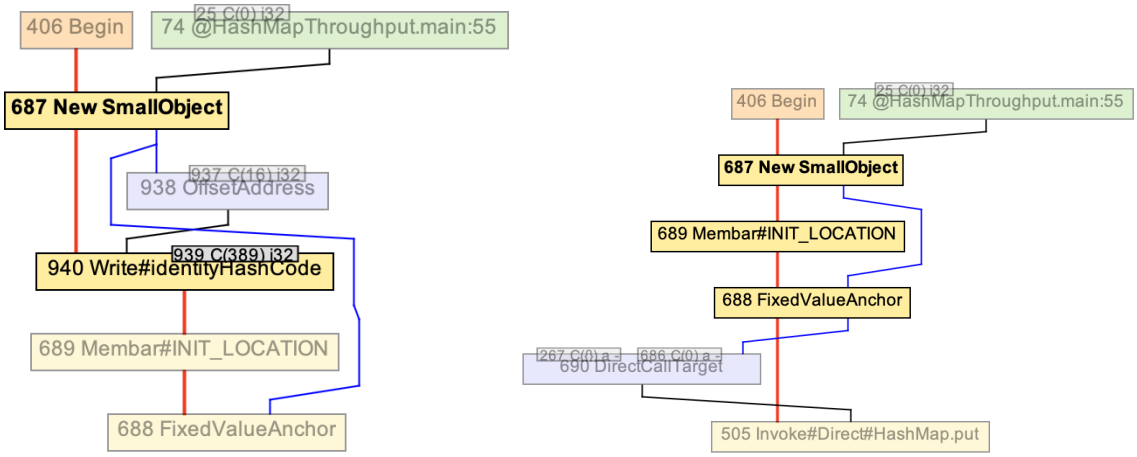
5.1.3 Project Repository

The repository containing the changes implemented during the course of this thesis can be found on https://github.com/lionas32/graal.

The files with the most important changes are as follows.[2]

- `FixedObjectLifetimeTable.java` serves as our interface to the OLD table. It provides all of the necessary methods to interact with the OLD table and retrieve and set object target generation.
- `SubstrateAllocationSnippets.java` and `AllocationSnippets.java` for most of the changes related to allocation. Some are presented in Listing 1 and Listing 2.
- `Space.java` provides the methods and necessary variables related to spaces and object promotions. Changes made are related to object promotion (see Listing 2).
- `ThreadLocalAllocation.java` for the changes made to introduce a second TLAB for the old generation.
- `HeapPolicy.java` and `CollectionPolicy.java` for the necessary changes to the GC policy.
- `GCImpl.java` for changes related to the caching and clearing of the OLD table.
- `InsertAllocationSitePhase.java` to instrument the insertion of an allocation site to the objects.
- `DecrementAgeVisitor.java` for tracking of deallocated objects.

[2]All of the files are under the `/svm` folder. Only the `InsertAllocationSitePhase` exists under `/compiler`.



(a) With a custom compilation phase. (b) Regular compilation.

Figure 5.1: Compilation graphs with (a) `InsertAllocationSitePhase` and (b) without (regular compilation).

- `IdentityHashCodeSupport.java` and `JavaLangSubstitutions.java` to manage identity hashcode behaviour with ROLP.

Changes to any other files contain only minor modifications in regards to enabling object pretenuring, or only serve the purpose of logging metrics.

5.2 Adding a Custom Compilation Phase

As mentioned in Section 4.2.1, to enable object lifetime profiling we have to embed a 30-bit allocation site value into the object layout, along with 2 bits for age. In our implementation, this is done through a custom compilation phase we have added in, known as `InsertAllocationSitePhase` (as noted above).

We apply the algorithm seen in Algorithm 1 to every `NewInstanceNode` and `NewArrayNode` in the compilation graph. These nodes hold information related to class and allocation site for objects.

The result from instrumenting the compiler to install a personal allocation site for a custom application containing object `SmallObject` can be seen in the Figure 5.1a. The sub-graph displays how the `InsertAllocationSitePhase` has modified the `New SmallObject` node with ID 687, representing the object allocation. We instruct the compiler that we perform a write-operation to the location within the object corresponding to the identity hashcode. This can bee seen through the node `WriteNode#identityHashCode` with ID 940 in Figure 5.1a. The offset to the identity hashcode for the object is contained within the `OffsetAddress` node (ID 938), and passed along to the `WriteNode#identityHashCode`. The

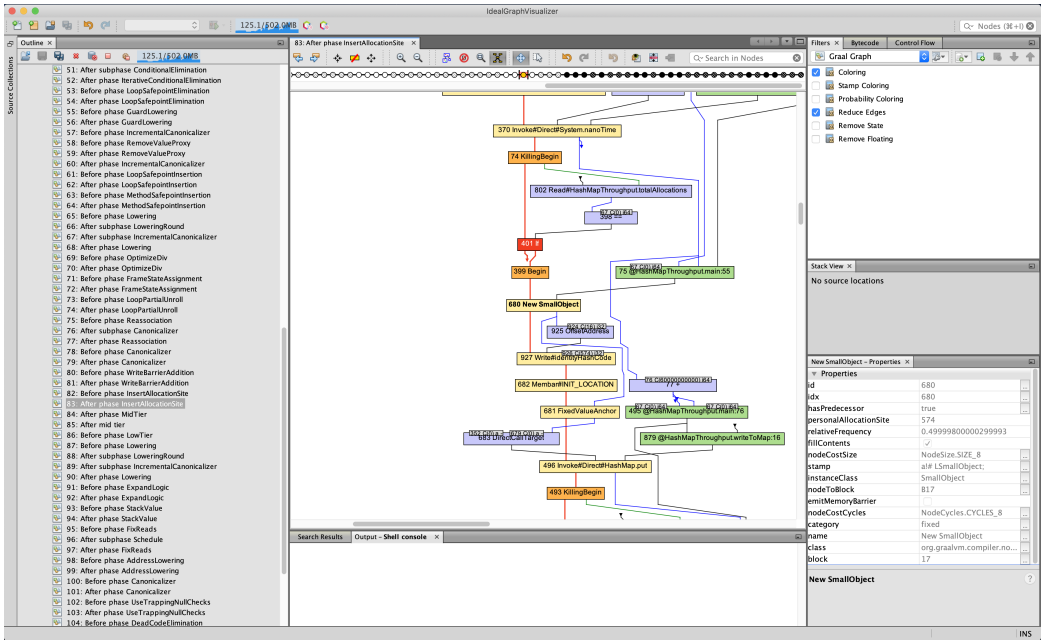


Figure 5.2: Ideal Graph Visualizer.

results of regular compilation without `InsertAllocationSitePhase` can be seen in Figure 5.1b. In this case, there is no `OffsetAddressNode` and `WriteNode` modifying the internals of the `New SmallObject` node.

5.2.1 Ideal Graph Visualizer

Ideal Graph Visualizer (IGV) [40] is a tool developed to visualize the compilation graphs of JVM programs. It is integrated into Native Image, and we use it during the implementation to confirm that the nodes responsible for the allocation site ID are generated and inserted correctly in to the compilation graph. A zoomed out version of the IGV compilation graphs in Figure 5.1 can be seen in Figure 5.2. The outline contained within the left-side contains the different compilation phases in the order they are processed. These phases can be selected, and we can see the resulting compilation graph of a selected phase in the main window.

5.3 Implementing OldTLAB

We introduce the second TLAB `oldTLAB` in the predefined class `ThreadLocalAllocation`. This is where most behaviour concerning allocation of new heap chunks, setting up of TLAB and slow-path allocation occur. A majority of the methods in the class are rewritten to allow allocation to happen both through the `youngTLAB` and `oldTLAB`, instead of just `youngTLAB`.

The most significant changes found in the class are included in the methods `slowPathNewInstance`, `retireToSpace` and `prepareNewAllocationChunk`. A short description of the methods and their modifications follows.

- `slowPathNewInstance` is the entry method for slow-path allocation (see Section 5.4.2 for how an allocation looks like). It leads to a fairly complex chain of behaviour related to allocating and setting up of new heap chunks for a TLAB. Our modification to the method includes diverting the object slow-path allocation to the `oldTLAB`, if its target generation is old. The necessary change to the method itself to achieve this is not large, but it requires that the chain of methods it calls is rewritten to allocate in the correct TLAB and achieve correct behaviour.
- `retireToSpace` is triggered right before a collection occurs. The method flushes all of the chunks contained within the TLABs to their corresponding spaces. In addition, it sets up the FOT for the objects in which the target generation is old (see Section 5.4.2).
- `prepareNewAllocationChunk` is the method responsible for allocating new heap chunks (or retrieving from a list of unused chunks) for a TLAB. The modification we have introduced is related to the counting of heap chunks which have been allocated in the old generation. The value is then used to trigger collections based on how many heap chunks are in the old generation. Modifications to the collection policy are covered in Section 5.5.2.

5.4 Object Lifetime Distribution Table

We have previously presented the main two events triggering changes in the OLD table: i) object allocation, and, ii) object promotion in Section 4.2.5. We start of by briefly explaining the implementation of the OLD table, before going in depth on the events triggering updates in the table.

5.4.1 Implementation of the OLD table

The Object Life Distribution table is implemented as a static hash-table. The size of the table is set to a pre-initialized static size of 6553. This size seems to be enough to cover all of the allocation sites, in addition to the allocation sites contained within the runtime code. The key values for accessing the table are integers corresponding to allocation site values of objects. All of the methods which interact with the OLD table make sure to mask out the lower 30 bits of the allocation site value before indexing lifetimes. This is because age occupies the 2 left-most bits (see

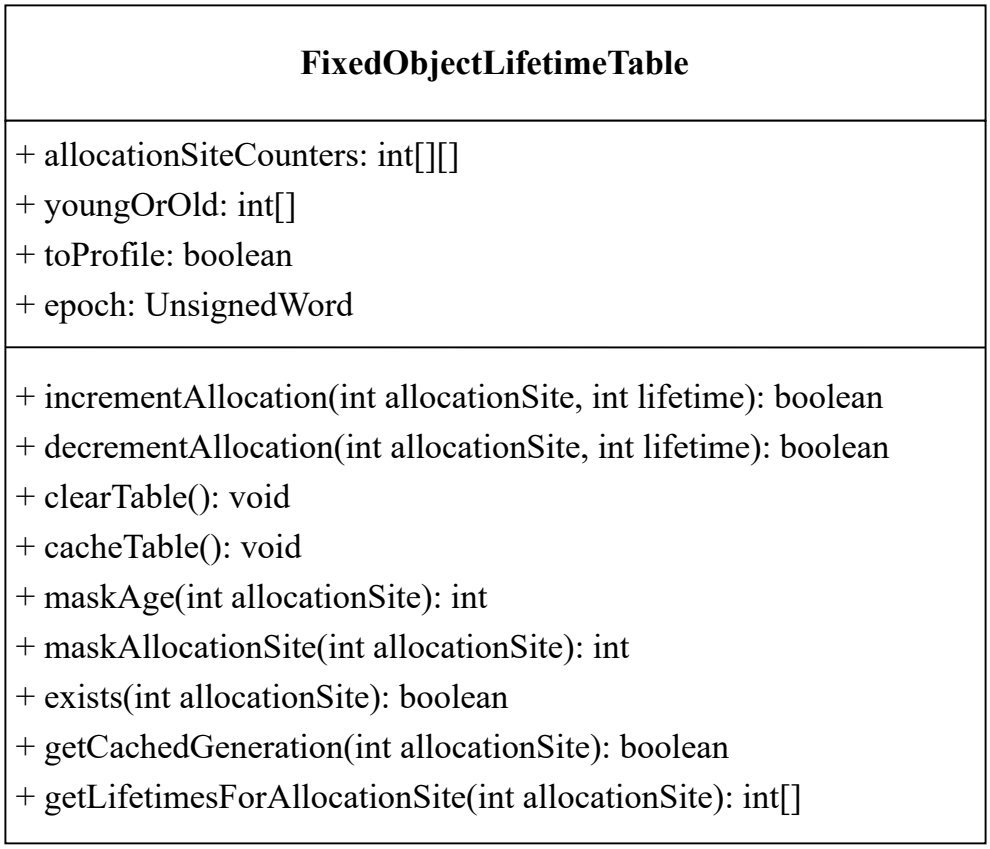


Figure 5.3: The fields of the `FixedObjectLifetimeTable` class.

Section 4.2.1). The entries of the OLD table are 4-length integer arrays, depicting lifetime statistics for a given allocation site. The 0th index corresponds to the amount of objects of age 0 for a given allocation site. All of the following indexes represent how many GC cycles an object has survived. Objects of age 3 (we count from 0), are at the maximum possible age and therefore we do not increment the amount of max age objects, if the object was at max age before promotion.

Since the allocation sites are generated as unique values bounded up to the OLD table size we can index the OLD table directly using the allocation site values. The methods implemented for the OLD table can be seen in Figure 5.3.

A short description of each of the methods which allow interaction with the old table follow.

- `incrementLifetime` to increment the amount of objects with `allocationSite` and `age`. Values updated in the `allocationSiteCounters` array.
- `decrementLifetime` to decrement the amount of objects with `allocationSite` and `age`. Values updated in the `allocationSiteCounters`

array.

- `cacheTable` and `clearTable`, which are called upon when we turn off profiling. We store the pretenuring information in the `youngOrOld` array (1 indicating target generation is old, 0 indicating young), and clear the `allocationSiteCounters` lifetimes array afterwards.
- `maskAge` and `maskAllocationSite` to mask out the age (2-leftmost bits) and allocation site (30-right most bits) from an integer.
- `exists` to check if `allocationSite` is a valid allocation site. An invalid allocation site would be of value 0 or above the maximum size of the `allocationSiteCounter` array.
- `getLifetimesForAllocationSite` to retrieve lifetime distribution of `allocationSite`.
- `getCachedGeneration` to retrieve if the target generation for `allocationSite` is old or young. Target generation is retrieved from the `youngOrOld` array.

Since all of the allocation site values are unique, the time complexity for all of the methods which index the table based on `allocationSite` are constant. Subsequent sections describe usages of the aforementioned methods.

5.4.2 Object Allocation

```
Object allocateInstance(hub, profilingData, ...){
    DynamicHub checkedHub = checkHub(hub);
    int site = profilingDatza.allocationSite()
    boolean forOld = false;
    if(epoch > OLD.stopProfilingEpoch){
        forOld = OLD.placeInOld(site)
    }
    Object result;
    if (forOld) {
        result = allocateInstanceImplForOld(encodeWithRememberedSet(hub),
                    profilingData, ...)
    } else {
        result = allocateInstanceImpl(encodeHeader(hub),
                    profilingData, ...)
    }
    return result;
}
```

Listing 1: A snippet demonstrating how we take a decision for target generation of an object.

```
Object allocateInstanceForOldImpl(object, profilingData, ...) {
    Object result;
    Word tlab = getOldTLAB();
    Word top = readTlabTop(tlab);
    Word end = readTlabEnd(tlab);
    Word newTop = top.add(size);
    if (useTLAB() && newTop.belowOrEqual(end))) {
        writeTlabTop(tlab, newTop);
        result = formatObject(object, ...);
    } else {
        result = callNewInstanceStub(object, true);
    }
    profileAllocation(profilingData, size);
    return result;
}
```

Listing 2: A snippet demonstrating bump-pointer allocation and subsequent increment of lifetime.

During the lowering[3] of the `NewInstanceNode` and `NewArrayNode` nodes, we pass along the generated allocation site to the profiling data which we can access upon corresponding object allocation. This is also why we also require that the applications are generated using `-H:+AllocationProfiling`. Otherwise, there would be no profiling data attached to each object type. The profiling data is represented by the `profilingData` parameter in Listing 1.

When the method responsible for allocation (see Listing 1) is triggered, a decision between placing the object in a chunk designated for the young or the old generation is taken. The variable `checkedHub` contains class information related to the object. The allocation site which allows us to do lookup of the target generation is currently located within the `profilingData`. We do not perform the target generation lookup until the epoch value is above `stopProfilingEpoch`, since we want a fair distribution of object lifetimes to build up. After profiling of objects has ended and all of the target generations are cached, we can use the target generation to allocate objects in either a heap chunk designated for the young or old generation. If the target generation is old, we allocate the object through `allocateInstanceImplForOld`. Otherwise, the object is allocated through the unmodified method `allocateInstanceImpl`.

The method `allocateInstanceForOldImpl` in Listing 2 is responsible for allocating an object in the old generation. It allocates objects through the *OldTLAB*, which is fetched through the method call `getOLDTlab`. The

[3]Lowering is the conversion of a higher-level abstraction to a lower-level. In this case it is the lowering of nodes representing allocations into methods responsible for allocations of actual object instances.

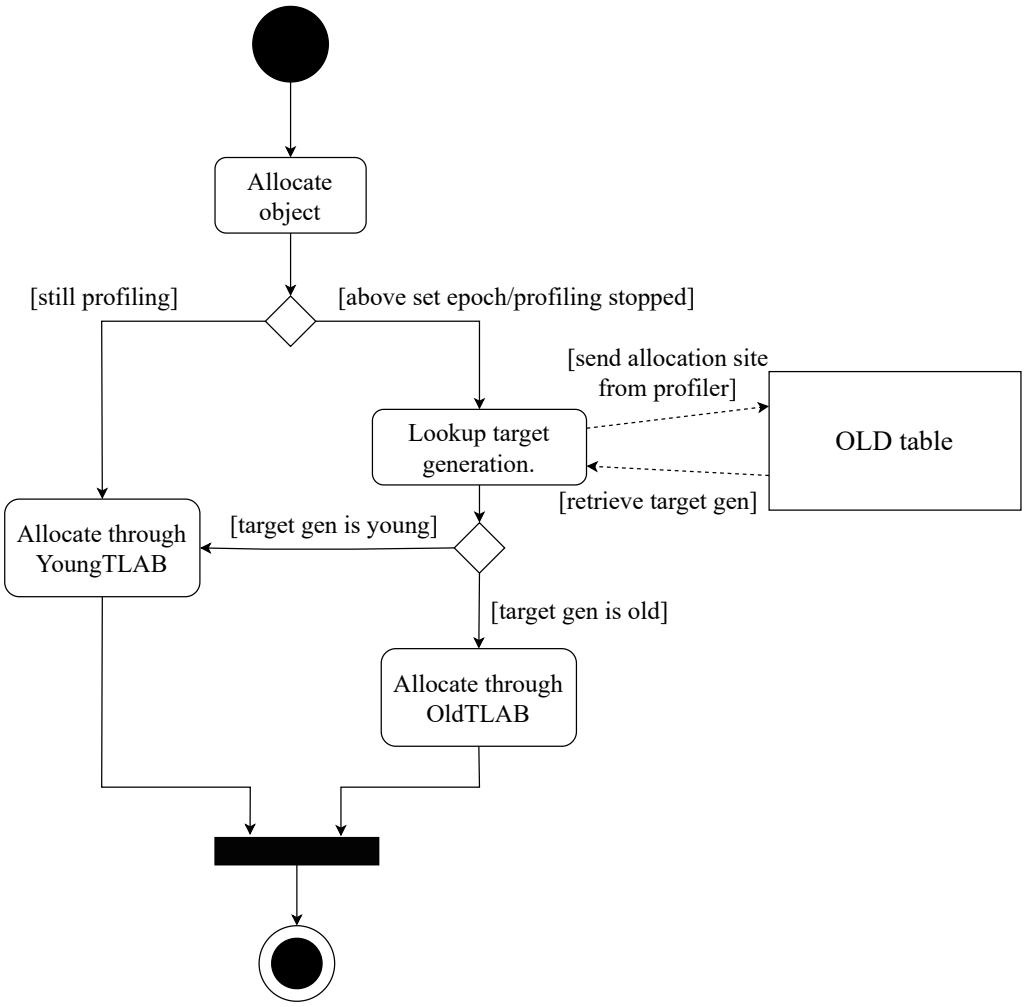


Figure 5.4: Object allocation with ROLP.

objects are then allocated either through the fast-path or the slow-path.

The separation between fast and slow-path can be seen in the two different branches of the if-statement in Listing 2. We run through the fast-path in the case the if-statement condition holds true. This is simply if the new top of the TLAB allocation chunk is below the end of the chunk. The fast-path is executed as bump-pointer allocation. This means we check that the allocation will not exceed the size of the chunk. If it does not exceed it, we proceed with aligning the object after the last allocated object in the chunk.

In the case in which the condition evaluates to false, we go through the slow-path. The call to `callNewInstanceStub` will either allocate a new heap chunk, or fetch one from a list of unused chunks. Which TLAB to allocate the heap chunk for is determined by the second parameter of `callNewInstanceStub`. In this case it is `true`, meaning create a heap chunk for the TLAB attached to the old generation. After a new heap

chunk is in place, bump-pointer allocation for the object is executed within `callNewInstanceStub`. The call to `profileAllocation` will increment objects of age 0 for a given allocation site in the OLD table. The allocation site is contained within `profilingData`, being passed along to `profileAllocation`.

It it is important to note that the code snippets above have been simplified from the actual implementation for the sake of example. The methods accept several other parameters related to type information, size of object and alike. We have implemented equivalent methods for array allocation which we do not feel is necessary to cover here. The main distinguishing difference is that array size effects the allocation, potentially putting an array in a unaligned chunk. In that case, we do not profile the array allocated, due to rareness of these types of allocations.

Figure 5.4 depicts in a more general sense how object allocation functions according to the aforementioned modifications.

Setting the First Object Table

We briefly mentioned how the card remembered set and FOTs works in Section 2.5.5. Therefore, when allocating objects in which the targeting generation is the old one, we have to set the remembered set bit for them. This modification can be seen in Listing 2, through the method call `encodeWithRememberedSet(checkedHub)`. The object header in this case is set with a `REMEMBERED_BIT`.

```
void setUpFirstObjectTableEntry(AlignedHeader that, Object obj) {
        Pointer fotStart = getFirstObjectTableStart(that);
        Pointer memoryStart = getObjectsStart(that);
        Pointer objStart = Word.objectToUntrackedPointer(obj);
        Pointer objEnd = LayoutEncoding.getObjectEnd(obj);
        setTableForObject(fotStart, memoryStart, objStart, objEnd);
}
```

Listing 3: Setup of the first object table for old generation objects.

In addition, we have to setup the first object table of each object contained within a old generation heap chunk. The code for it can be seen in Listing 3. The necessary offsets are calculated through builtin methods. The offsets are the location of the FOT within the heap chunk, object start within the chunk, and object pointers to start and end of the object we are setting the FOT for. This is done during the retiring[4] of the heap chunks right before a collection. Since it is done for each object within heap chunks about to be retired to the old from-space, it does induce a overhead. It

[4] Appending of heap chunk to its corresponding allocation space.

```
Object promoteAlignedObject(Object original, Space originalSpace) {
    int allocationContext = 0;
    if(SubstrateOptions.RolpGC.getValue() && OLD.toProfile){
        allocationContext = computeLifetimeBeforePromotion(original);
    }
    ...
}

int computeLifetimeBeforePromotion(Object obj){
    int hashCodeOffset = getHashCodeOffset(obj);
    int allocationContext = readInt(obj, hashCodeOffset);
    if (allocationContext == 0 || OLD.exists(allocationContext)) {
        return 0;
    }
    int prevAge = getAge(allocationContext);
    int newAllocationContext = incrementAge(obj, allocationContext);
    int newAge;
    if (newAllocationContext == allocationContext) {
        newAge = prevAge; // object already at max age
    } else {
        newAge = prevAge + 1;
    }
    if(newAge > 0 && newAge != 0b11 || prevAge == 0b10) {
        boolean inc = OLD.incAllocation(allocationContext, newAge);
        boolean dec = OLD.decAllocation(allocationContext, newAge - 1);
    }
    return newAllocationContext;
}
```

Listing 4: Computing of lifetime during promotion.

seems to be necessary to perform this for each object designated for the old generation, since they can contain references to objects in the young generation. We set the FOT during retiring of chunks since this is when the application state is at a safepoint.[5] We cover this here since we will see in Chapter 6 that this can affect collection times with ROLP.

5.4.3 Object Promotion

When a collection is triggered all of the reachable objects in the from-spaces (eden space and old from-space) are moved into the old to-space, which then becomes the new old from-space. This copying of objects is the second major event responsible for inducing significant change to the distribution object lifetime table. The implementation of computing new

[5] A state of the program in which memory, registers and thread-local variables can be safely inspected.

lifetimes for objects can be seen in Listing 4.

The promotion of an object in an aligned chunk happens in the builtin method `promoteAlignedObject`. We exclude details related to copying since they are not relevant for the lifetime computation. The relevant part of the method is the if-statement in which we check if profiling is still ongoing (depends on the value of `OLD.toProfile` variable). If profiling is ongoing, we compute new age for objects and update the OLD table through the method `computeLifetimeBeforePromotion`.

There are two expensive read/writes to memory that occur during `computeLifetimeBeforePromotion`. The first one is when we need to read the allocation site of the object. This can be seen in the call to `readInt`, which uses the object hashcode offset to fetch the allocation site. The age value of the object is contained in the 2 left-most bits of the site value. If the age value is not at the maximum (3), we increment the previous age value and write it to the object. These two read and write operation are the main source of overhead during the profiling. The write operation is avoided if the object is at maximum allowed age.

After the age computation, we determine if we should change the distribution within the OLD table. We simply check if the object was at max age before. If it was, we do not change the distribution. Otherwise, we decrement the amount of objects with the previous age and we increment the amount of objects with the new age. The change in the distribution occurs during the `OLD.incAllocation(...)` and `OLD.decAllocation(...)` calls.

The method `promoteAlignedObject` also highlights the usage of the `RolpGC` option. It is prevalent in all of the other methods in which we have to hook into with our profiling and pretenuring code. We have included it here to just show how we separate the behaviour from the case in which ROLP is not enabled, since code within the option is only executed if the user has requested it.

5.4.4 Caching of Target Generation

The lifetime distributions for each allocation site are cached and cleared upon a pre-defined epoch (default value of 16). This value is contained within the `FinalEpoch` variable in Listing 5, used in the `collectOperation` method. This method is responsible for performing collections, which we exclude details of. If the condition in the method is true, we start of by caching the table.

The caching is performed in the `cacheTable` method, located within `FixedObjectLifetimeTable.java`. We go through each allocation site, represented by each index in the `allocationSiteCounters` array. Count-

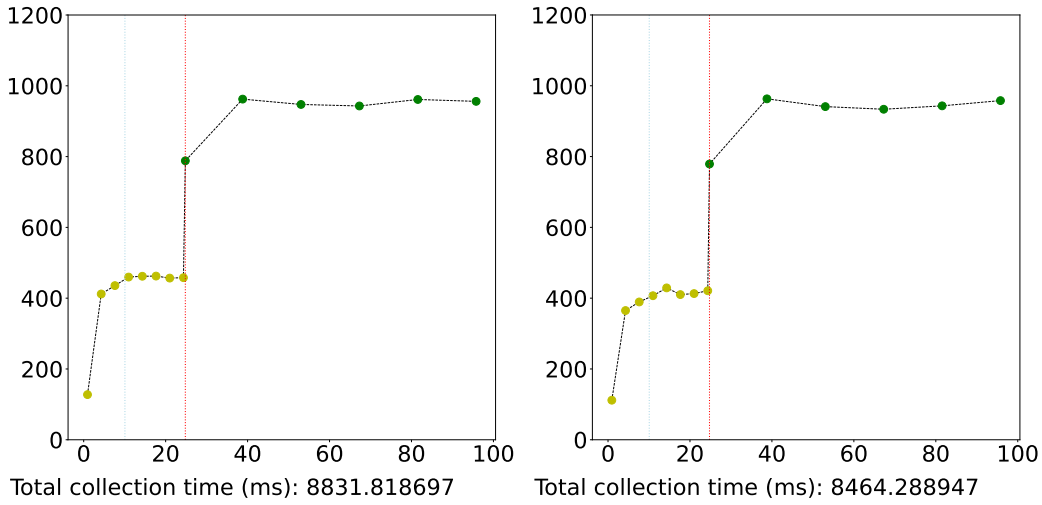
```
// Contained withe GCImpl.java
boolean collectOperation(GCCause cause, UnsignedWord requestingEpoch) {
    ... // functionality related performing a collection
    if(FixedObjectLifetimeTable.toProfile && collectionEpoch.equal(FinalEpoch)){
        FixedObjectLifetimeTable.cacheTable();
        FixedObjectLifetimeTable.clearTable();
    }
    ... // functionality related to cleaning up after collection
}

// Contained within FixedObjectLifetimeTable.java
static void cacheTable(){
    for(int i = 1; i < STATIC_SIZE; i++){
        int[] allocs = allocationSiteCounters[i];
        boolean toCache = allocs[0] < allocs[1] + allocs[2] + allocs[3];
        if(toCache) {
            if(youngOrOld[i] == 0){
                youngOrOld[i] = 1;
            } else {
                // this case skipped
            }
        }
    }
    toProfile = false;
}
```

Listing 5: Caching of target generation.

ing starts from 1, since we use 0 as a invalid allocation site value. We cache the target generation for each allocation site by evaluating the condition `allocs[0] < allocs[1] + allocs[2] + allocs[3]`. The intial index `allocs[0]` contains amount of objects which did not get promoted. `allocs[1-3]` contain the corresponding amount of objects with lifetimes ranging from 1 to 3. If the condition is true, it means that the objects for the current allocation site are long-lived, and thus we cache a 1 in the `youngOrOld` array to indicate that. The value we set in the array is otherwise a 0, indicating allocate the objects in the young generation.

At the end of the caching we disable profiling. This is done by setting `toProfile = false`. Instead of profiling, the values within the `youngOrOld` array will now be utilized to allocate objects in either the young or old generation. After the caching of table, `clearTable()` clears the `allocationSiteCounters` array of lifetime information.



(a) With `DecrementAgeVisitor` enabled. (b) Regular ROLP.

Figure 5.5: Collection times with `DecrementAgeVisitor` enabled (a) and without (b).

5.4.5 Tracking consumed chunks

When a collection happens, all of the unused chunks are either free'd to the operating system or added to a free list of unused chunks. This is done through a builtin method `consumeAlignedChunk`, which is called after each collection. By walking through the dead chunk passed to the method, we can identify objects which are not forwarded (promoted to the old to-space), and simply decrement their corresponding lifetime. Objects which are not forwarded are identified if they do not contain a forwarded bit in their header.

If one wants to gather more accurate lifetime statistics, one can enable this in our implementation. We have implemented a custom `DecrementAgeVisitor`. If enabled, this visitor object is called on each heap chunk passed to the `consumeAlignedChunk` method. It runs through each object in the heap chunk. If the object is forwarded, we do not visit it. This is because there exists a live copy of it in the to-space. Otherwise, we lookup the object allocation site and decrement the age of it in the OLD table.

The problem which arises from the technique mentioned, is a noticeable increase in overhead during profiling. This is due to the fact that we are required to a read operation on each object in the heap chunk to determine if the object was promoted or not. We can see this in a simple native image application in which we write to a cache in round-robin fashion in Figure 5.5. In the figure we have collection times with and without `DecrementAgeVisitor` enabled. During the profiling phase (before the

red indicator), Figure 5.5a has about a 11% increase in total collection time compared to Figure 5.5b. In addition, we can see that for this example the collection times after the profiling remain the same. This means that the pretenuring information is equal for all of the objects in the evaluations above, thus indicating that enabling `DecrementAgeVisitor` does not improve the pretenuring of objects in this case.

The reason for the sudden increase in individual collection times for both evaluations after profiling, is due to the fact that all objects are identified as being long-lived and allocated directly in the old generation. This in turn triggers only complete collections.

To enable deallocated object tracking, one can uncomment lines 162-164 in `DecrementAgeVistor.java`.

5.5 Preserving Necessary Functionality

Some of the changes that we introduce through ROLP require modifications to the behaviour of the runtime such that we do not impend on standard functionality. These are mainly related to managing of object identity hashcode and maintaining a fair collection policy. The implementation details of these modifications are presented in the upcoming sections.

5.5.1 Managing Object Identity Hashcode with ROLP

The allocation site of objects occupy the object field meant for identity hashcode. Initially, all objects hold a value of 0 in the hashcode field. Therefore, overwriting this with the allocation site does not cause any issues initially. However, the identity hashcode for objects is generated during the first call of `obj.hashCode()` or `System.identityHashCode(obj)`[6] (or any other builtin means of retrieving object hashcode). If this call occurs for an object and it contains an allocation site ID, we now need to overwrite it and exchange it with a valid identity hashcode value.

If the hashcode field within the object contains an allocation site value, `overwriteContextForHashCode` executes for that object. This can be seen in Figure 5.6, occurring after a call to `obj.hashCode()`. The method initially decrements the amount of objects with its corresponding allocation site. Subsequently, a valid hashcode value is generated through a builtin method `generateHashCode`. The method has been modified to initialize hashcode values to be between the size of the table and the maximum allowed integer value in java[7]. This is to avoid conflicts with allocation site values, which all are within the range of the OLD table size.

[6] If the class of the object has not overwritten the hashcode method `.hashCode()`.
[7] $2^{31} - 1 = 2,147,483,647$

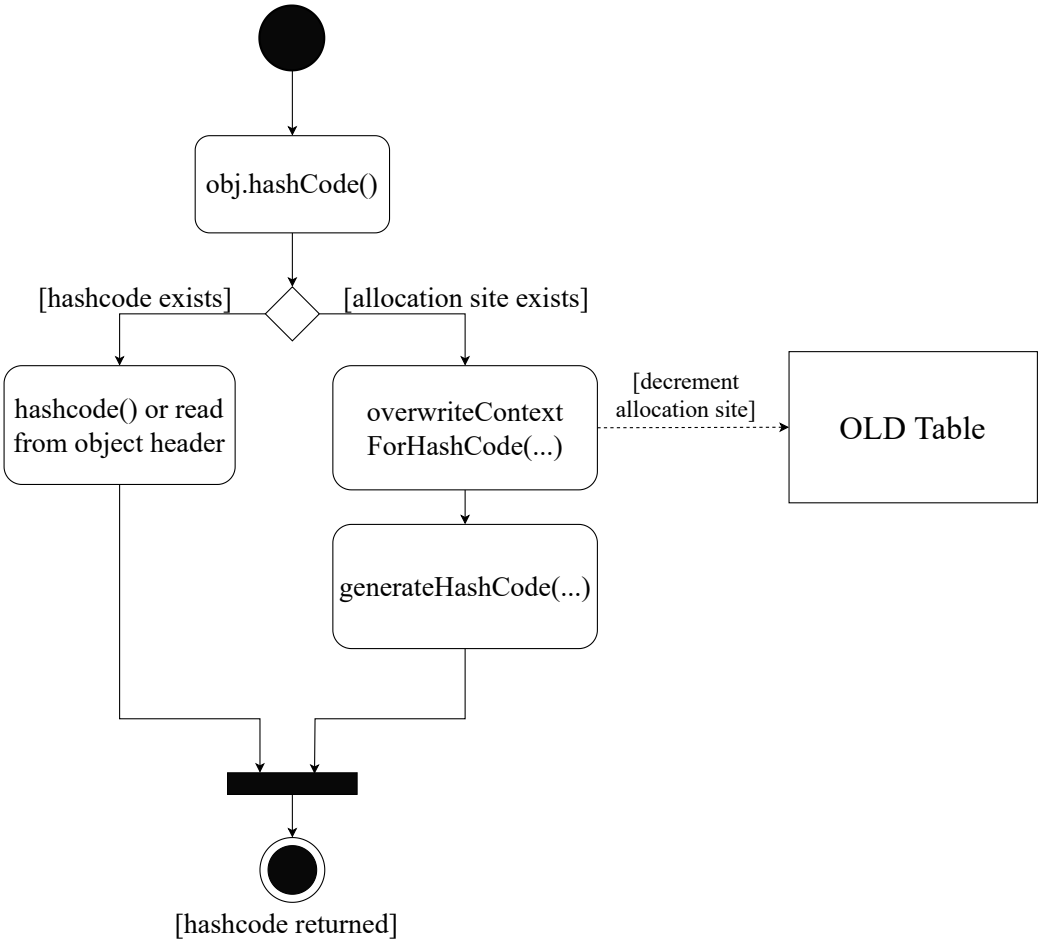


Figure 5.6: Activity diagram displaying management of object hashcode.

After generating the hashcode value, we then utilize the `Unsafe` class to overwrite the object hashcode field in memory, which now instead of containing the allocation site value for the object will have a valid identity hashcode value embedded.

An activity diagram is depicted in Figure 5.6 describing the flow of generating a valid hashcode. If the `.hashcode()` method is overridden, there no need to overwrite the hashcode field in the object layout (the left-path of the diagram). Otherwise, the object most likely contains an allocation site value. In that case, overwrite it with a hashcode value and return it as an integer.

5.5.2 New Collection Policy

Using ROLP requires a collection policy which takes regard for objects allocated in the old generation. We implement this through a custom policy `ByTimeWithRolp`. All collection policies have to contain two methods, `collectIncrementally` and `collectCompletely`. These can be

```
// Methods contained within ByTimeWithRolp.java
@Override
public boolean collectIncrementally() {
    return true;
    }

@Override
public boolean collectCompletely() {
    boolean result = collectCompletelyBasedOnUsedBytes(trace);

    return result;
}

private static boolean collectCompletelyBasedOnUsedBytes(Log trace) {
    UnsignedWord withFullPromotion = HeapPolicy
        .withFullPromotion; // Threshold value for complete collection

    UnsignedWord oldInUse = HeapPolicy.getOldUsedBytes();
    return oldInUse.aboveOrEqual(withFullPromotion);
}
```

Listing 6: The methods of the collection policy `ByTimeWithRolp`

seen in Listing 6. For all policies, these methods are called when the GC has to decide if it will perform an incremental or a complete collection. The behaviour for these methods defined through our policy is as follows:

- `collectIncrementally` always evaluates to `true`. This is because the method is called if a collection was decided to occur due to a slow-path allocation, which is independent of the method above, but relates to the occupied size of the young generation. This is also how Serial GC by default implements incremental collections.
- `collectCompletely` trigger a complete collection, if the occupied space in the old generation is beyond a limit. We define this limit as the maximum heap size subtracted twice with the young generation size. The reason for subtracting twice is that a complete collection has room for a promotion of the young generation. This limit value is contained within the variable `withFullPromotion`.

Throughout the implementation, an option to only trigger complete collections without incremental collections was explored. All complete collections entail incremental collections by default. Since a young generation in ROLP might potentially not be full when triggering a complete collection, we could trigger only a complete collection scan of the old generation. The changes for this feature look like following:

Experimentation with this feature looks promising, but it seems that

```
public boolean collectIncrementally() {
    return getYoungUsedBytes().aboveOrEqual(getMaximumYoungGenerationSize());
}
```

Listing 7: Modified incremental collection condition.

there might be difficulties which can potentially arise in more complex applications, such as live objects becoming unreachable. Thus, all of our complete collections in the current version entail also incremental ones.

To enable the ROLP collection policy, one has to specify `ByTimeWithRolp` as the initial collection policy. If one wants to modify the limit value `withFullPromotion` to be a certain percentage of the maximum heap size, the option `PercentHeapThreshold` can be adjusted to allow for it.

5.6 Summary

In this chapter, we have gone through our integration of ROLP into Serial GC. We described in detail the profiling techniques we use, modifications to the heap model, and how the OLD table is managed and utilized for pretenuring.

In the next chapter, we evaluate our implementation and measure its performance in terms of throughput and latency in comparison to regular Native Image.

Part III

Evaluation and Conclusion with Future Work.

Chapter 6

Evaluation

In this chapter we present the results from the different experiments performed for ROLP together with SubstrateVM. We initially present the setup of the hardware and software we perform the evaluations on. Afterwards, we introduce the experiments we will be evaluating. Then we go through the results of the experiments.

Throughout the chapter, we refer to the modified version of native image simply as **ROLP**, whilst the unmodified native image version is referred to as **Vanilla**.

6.1 Setup

In this section we provide an overview over the hardware which been used for the experiments, as well as the versions of software modified and utilized. Then, we present the most relevant runtime and build options which have been set and modified for the experiments and give a brief overview over the experiments we evaluate.

6.1.1 Specification

Type	Value
Machine	MacBook Pro (16-inch, 2019)
Operating System	macOS catalina 10.15.7
Processor	2,6 GHz 6-Core Intel Core i7
Memory	16 GB 2667 MHz DDR4
Storage	512 GB SSD

Table 6.1: Specifications of the machine we performs the tests on.

Type	Value
GraalVM	OpenJDK 64-Bit Server VM GraalVM 20.3.0-dev
JVMCI	Build 25.262-b10-jvmci-20.20-b03, mixed mode
OpenJDK	Version 1.8.0_262

Table 6.2: Specifications of the GraalVM version and required JVM.

The specifications of the hardware on which we perform the evaluations can be seen in Table 6.1. The modified version of GraalVM and the utilized OpenJDK and JVMCI versions can be seen in Table 6.2.

6.1.2 Memory Utilization

Before we introduce the experiments, we present some of the build and runtime options relevant for the experiments. Most of these are at default values set by the runtime, unless otherwise stated. The options presented here differ from the ones present in section 5.1.2 in that they do not directly influence the behaviour of ROLP, and mostly are related to the behaviour of the runtime and memory utilization.

- Heap utilization is modified throughout our experiments. This is to increase the predictability of our experiments, since by default utilized heap size is set by sampling physical memory, which can affect the GC behaviour significantly if available physical memory is high. The most relevant options for heap utilization are:
  - ▷ **Maximum Heap Size** which is set by with `-R:MaxHeapSize` as a compilation option or `-Xmx` as a runtime option. By default uses of 80% of total physical memory. This is the only option which we directly modify for our applications.
  - ▷ **Minimum Heap Size**, which is initialized to be the double of the young generation size.
  - ▷ **Young Generation Size**, which by default is set to 10% of the maximum heap size (but not more than 268 MB).
- `AlignedChunkSize` at 1048.576 KB. This represents the size of a heap chunk. A collection can trigger if a new aligned chunk is allocated or required.
- `LargeArrayThreshHold` at 131.072 KB. Arrays above this size are allocated in a separate unaligned chunk, which we do not profile. This threshold is by default set to be at 1/8 of heap chunk size.

  For Serial GC both with and without ROLP, the time-based complete collections (see Section 4.1.2) have been **disabled**. This is due to

their unpredictability at which times they can occur. In addition, it makes it easier to compare the individual collection if they occur approximately at the same heap threshold. Collections during our evaluations only trigger if heap utilization reaches a threshold.

6.1.3 Experiments

There are two major applications which we evaluate, *Circular Array* and *Circular Hashmap*. Both of them utilize object allocation heavily and thus trigger the GC frequently. For all of the evaluations we present throughout the chapter, we have made sure to run the experiments for a long enough time for GC collection times to stabilize. We introduce the experiments in the following sections.

Circular Array

In this application objects are allocated into an array in a circular style at maximum throughput. This means that after the array is filled up, we start over from the first index. The writing to the array continues for a set amount of allocations. The options that are modifiable for the experiment are:

- **Amount of objects** - This is the total amount of objects allocated into the array throughout execution.
- **Array size** - Total size of the array. After the array is filled up, we start allocating from the beginning of it, thus overwriting old objects.
- **Read rate** - Reading from the array reads an object from a random index and performs some work on it. The work involves generation of short-lived objects.

Modifying the size of the array impacts the performance of the GC. If the array is relatively small compared to total heap size, objects are often overwritten. This means a majority of objects die before promotion occurs. If the size of the array is fairly large, newly allocated objects in the array survive incremental collection.

We have two major experiments for the *Circular Array* experiment. One which only involves writing to the array. In the second experiment, we enable reading from the array. The results are presented in correspondingly in Section 6.2.1 and Section 6.2.2. Condensed results for multiple array sizes is presented in Section 6.2.3.

Circular Hashmap

This application functions as a more complex version of the aforementioned *Circular Array*. Objects are allocated in the same away except that the data-structure utilized as our cache is the builtin `Hashmap` implementation of Java. We utilize a simplified version of the `Integer`-class as our key values, called `CustomInteger`. This is to prevent object generation from autoboxing/unboxing.[1] The options that are modifiable for the experiment are:

- **Amount of objects** - This is the total amount of objects allocated into the array throughout execution.
- **Key bound** - Since a hashmap is an unbounded data-structure, we need an option which indicates the "end" of the cache.
- **Read rate** - To simulate utilization of values contained within the cache. Produces short-lived objects.

When we reach the end of the cache, allocations into it begin anew from the lowest index (`CustomInteger` containing value 0), overwriting old values contained into the indexes we are inserting new objects.

In this experiment it is important to note that the hashmap produces a considerable amount of objects on its own, being a non-primitive data-structure. In addition, the keys it utilizes are also objects. This affects the performance of the GC substantially. This is covered throughout Section 6.2.4 and Section 6.2.5. Condensed results for multiple cache sizes is presented in Section 6.2.6.

6.2 Results

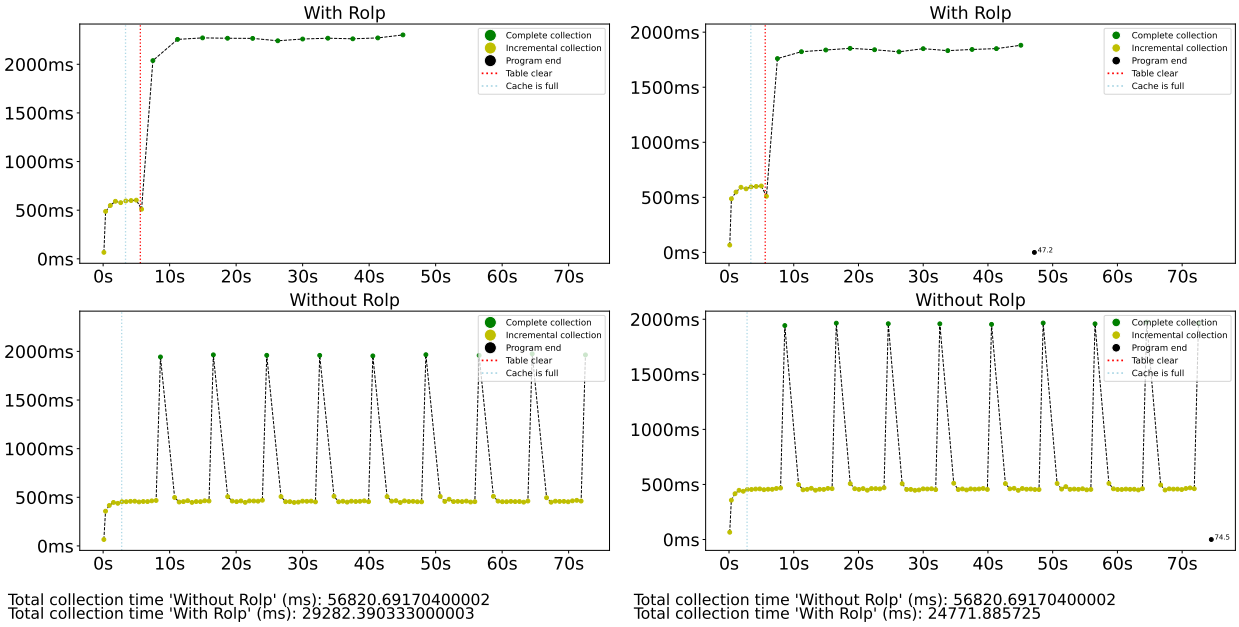
In this section we present the results of the different experiments for the applications covered throughout Section 6.1.3. For each of the experiments we start of by covering the setup, before going through the results for time spent performing collections, allocation time percentiles and overall throughput.

6.2.1 Circular Array

The options that we run the experiment with are:

- `-Xmx4g` for the maximum heap size. This results in a maximum heap of 4.3 GB and a young generation of 268 MB.
- 786 000 000 as total amount of allocations.

[1] https://docs.oracle.com/javase/tutorial/java/data/autoboxing.html



(a) Collection times.

(b) Collection times without FOT setup.

Figure 6.1: Collection times for *Circular Array*.

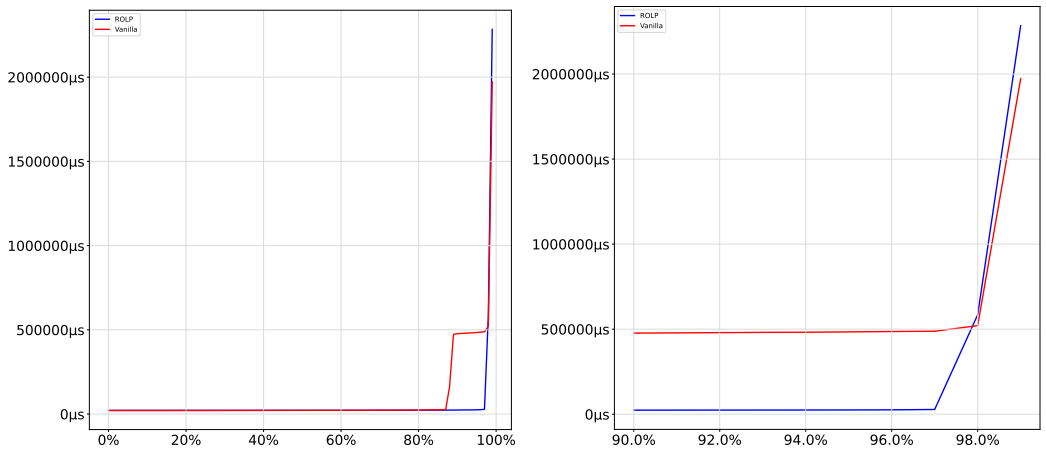
- 40 000 000 as cache size. This results in a fully-filled cache occupying 1.6 GB of the heap (37%).
- `-H:FinalEpoch=8` to dictate the final profiling epoch to be 8. This option is ROLP specific.

In this experiment, no reading from the array is performed.

Collection Times

We can see in Figure 6.1a that there is no incremental collections occurring after profiling compared to the Vanilla run. The reason is that ROLP has identified the majority of often allocated objects as long-lived. This is also why we only see complete collections occurring. By eliminating the need for incremental collections, we can see that the execution ends a significant amount earlier, 47.2 seconds with ROLP, and 74.5 seconds for Vanilla. This is the result of the 48% decrease in total time spent performing collections compared to Vanilla.

A noticeable feature of Figure 6.1a is the increase in complete collection time for ROLP. This the result of the need to setup the FOT (see Section 5.4.2). If we do not consider the overhead from FOT setup, the complete collection times are comparable to the run without ROLP, as seen in Figure 6.1b.



(a) 0-99 percentile.

(b) 90-99 percentile.

Figure 6.2: Allocation time percentiles. Each point is the percentile for the time it takes to allocate 1 000 000 objects.

Allocation Time Percentiles

If we look at the amount of time it takes to allocate the objects in Figure 6.2 we can see that there is a slight increase in time spent between the 0-90% percentiles for ROLP. The main reason for this is the need to make a decision for each object before between placing it in the young or old generation, which the Vanilla implementation avoids. However, we can see that the collections have a huge impact for the highest percentiles, with ROLP taking a significant time lower. This is because we eliminate a significant amount of incremental collections, resulting in reduced long-tail latencies. The worst percentiles outperform ROLP, due to slightly longer complete collection times.

Throughput

In the throughput plot in Figure 6.3 the allocations are grouped into 4 second intervals. This is because a collection can extend passed 2 seconds, and by grouping allocations into data-points of 4 seconds we can get a general overview of the allocation pattern throughout execution. In this plot we can see after the initial dip during profiling, ROLP achieves significantly higher throughput throughout the whole execution compared to vanilla.

6.2.2 Circular Array with Read

In this experiment we have enabled reading from the array. The options that we run the experiment with are:

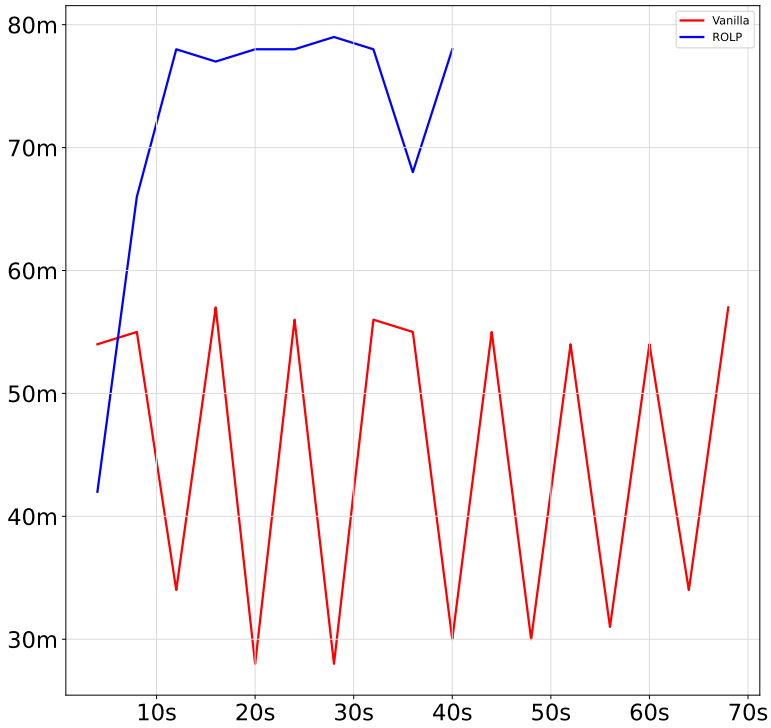


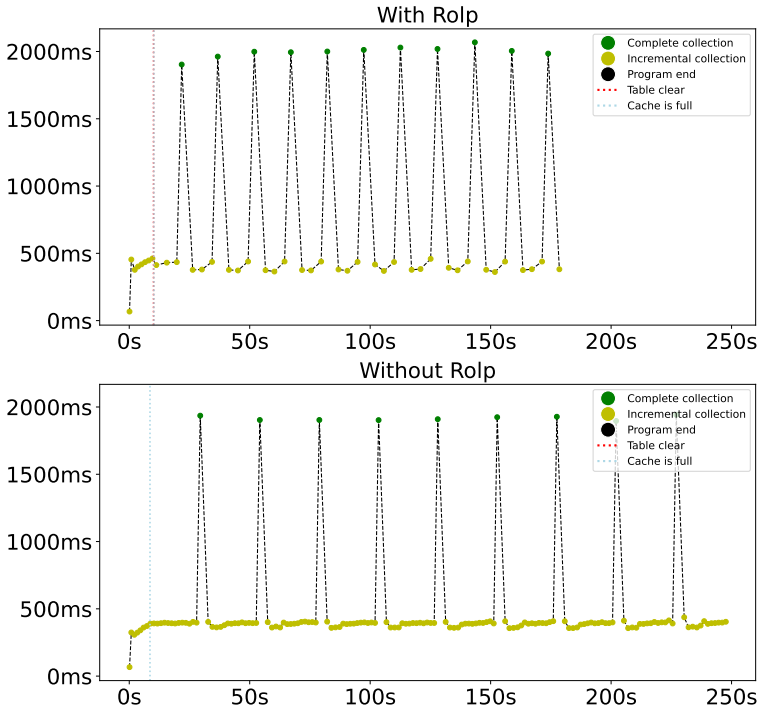
Figure 6.3: Throughput of *Circular Array*. The Y-axis depicts objects allocated. The X-axis depicts execution time. Each point contains objects allocated for intervals of 4 seconds.

- `-Xmx4g` for the maximum heap size. This results in a maximum heap of 4.3 GB and a young generation of 268 MB.
- 786 000 000 as total amount of allocations.
- 40 000 000 as cache size. This results in a fully-filled cache occupying 1.6 GB of the heap (37%).
- Read rate of 2 000 000 objects per second.
- `-H:FinalEpoch=8` to dictate the final profiling epoch to be 8. This option is ROLP specific.

The results from the experiment can be seen in pages 71-72. In Figure 6.4 we have a execution run from the *Circular Array* application in which reading from the array is enabled. Each read from the array produces a short-lived object, which is indicated by the incremental collections. The amount of incremental collections occurring is still substantially lower than Vanilla, with only 42 incremental collections occurring for ROLP, and 145 for Vanilla. This is because the only short-lived objects we have identified are the ones that are being produced by the reading of the array.

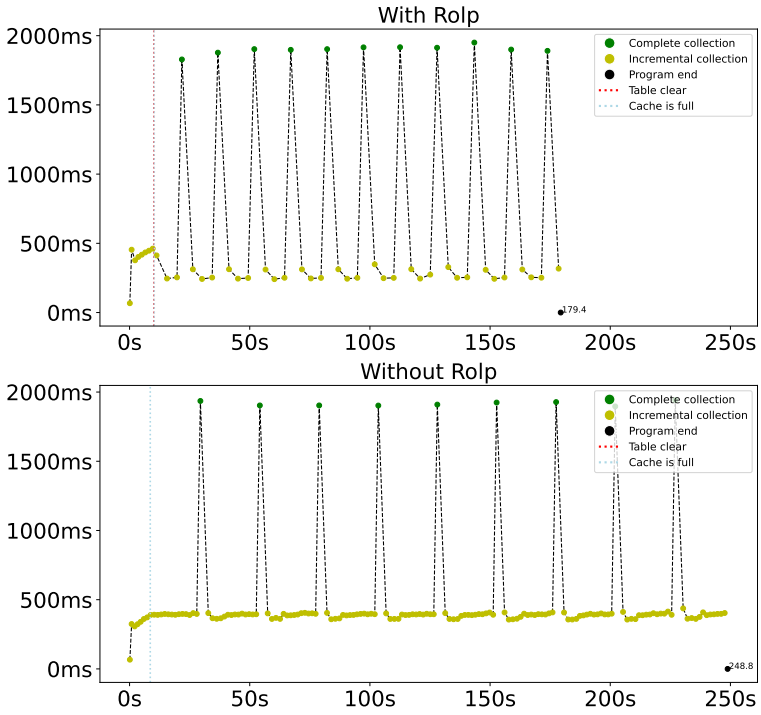
Regarding individual collection times, we can see a slight increase in the

time spent performing incremental collections in Figure 6.4. This is due to the fact that the incremental collections perform some of the work regarding setting up of the FOT.



Total collection time 'Without Rolp' (ms): 72971.29296800002
Total collection time 'With Rolp' (ms): 38651.22910999999

Figure 6.4: Collection times with reading.



Total collection time 'Without Rolp' (ms): 72971.29296800002
Total collection time 'With Rolp' (ms): 33354.903392

Figure 6.5: Collection times without FOT setup (with reading).

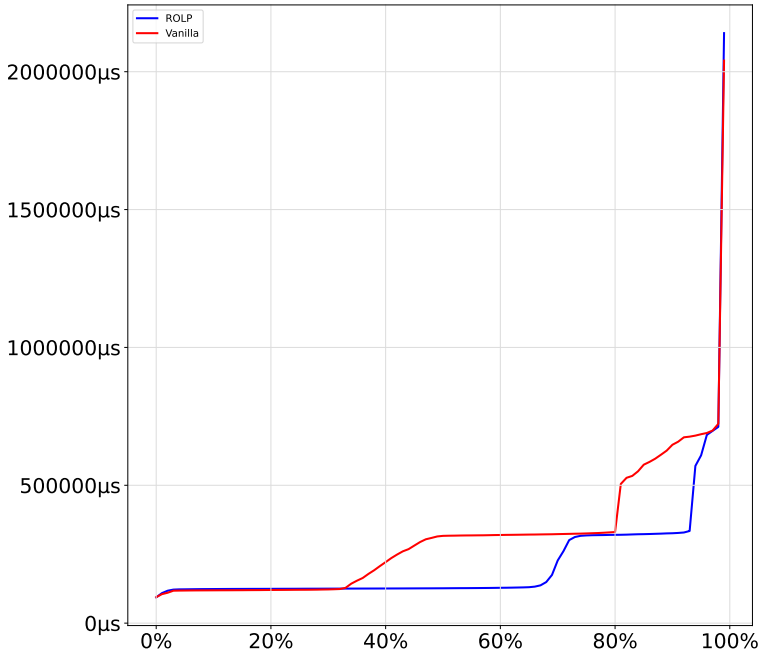


Figure 6.6: Allocation time percentiles for *Circular Array* with reading.

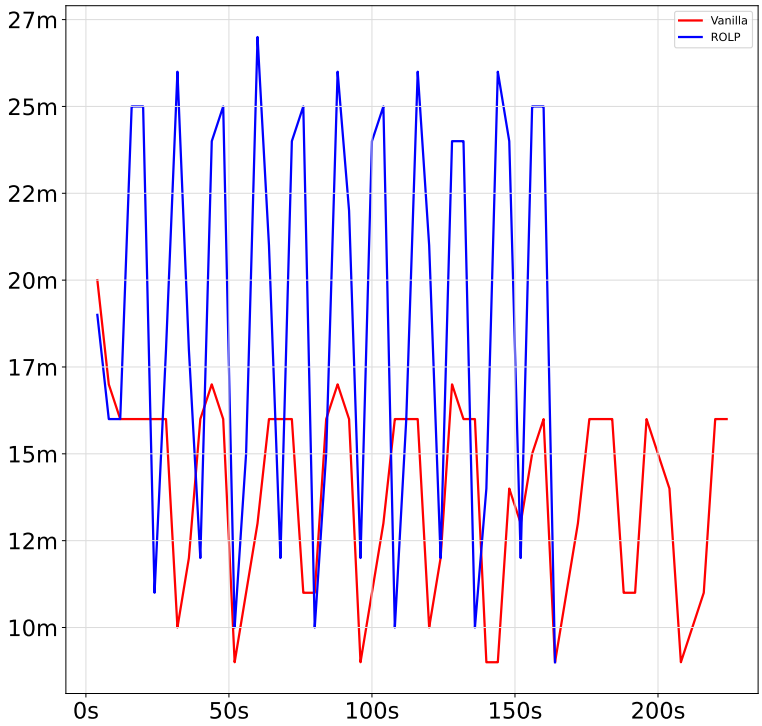


Figure 6.7: Throughput of *Circular Array* with reading. Each point contains objects allocated for intervals of 4 seconds.

6.2.3 Discussing Results from Circular Array

We provide condensed results from different *Circular Array* runs in tables in the following pages. What each of the columns represent is described in a left-to-right order corresponding to the table in the following listing:

- The **cache size** is the size of the array. The percentage value next to it describes the percentage of occupied heap size due to a full cache.
- Amount of incremental collections during the the whole execution.
- Amount of complete collections during the the whole execution.
- Total collection time.
- Total execution time.

In Table 6.3 and Table 6.4 the parameters for total amount of allocations and heap size are the same as the ones defined for the *Circular Array* run in Section 6.2.1. In the aforementioned tables we have strictly enabled writing without reading. In Table 6.5 and Table 6.6 we show results with enabled reading of 2 000 000 objects per second.

In all of the aforementioned tables, we can see that there is a significant difference between ROLP and vanilla in the amount incremental collections which occur, across different cache sizes. Avoiding these incremental collections with ROLP results in significantly lower collection times compared to Vanilla, which equivalently decreases the amount of total execution time for the same amount of work.

At 80 000 000 cache size, we can see that results for ROLP lag behind Vanilla in terms of total collection time. This is the result of utilizing 75% of the heap size, which triggers complete collections more rapidly, as there is less heap space. This is avoided to some degree by Vanilla because it utilizes in addition 268 MBs of the young generation. However, results with this cache size can be considered a special case, since increasing maximum heap size would benefit ROLP here.

Cache size	**Incremental**	**Complete**	**Collection time**	**Execution time**
10 000 000 (9%)	9	7	12.3s	30.0s
20 000 000 (19%)	9	8	16.9s	34.9s
40 000 000 (37%)	9	11	29.2s	47.2s
60 000 000 (56%)	9	17	55.6	73.5s
80 000 000 (75%)	9	41	153.5s	171.8s

Table 6.3: Statistics for ROLP for different *Circular Array* runs with only writing enabled.

Cache size	**Incremental**	**Complete**	**Collection time**	**Execution time**
10 000 000 (9%)	88	6	31.1s	48.6s
20 000 000 (19%)	87	7	39.2	56.9s
40 000 000 (37%)	86	9	56.8s	74.5s
60 000 000 (56%)	83	12	79.3	91.1s
80 000 000 (75%)	74	21	122.0s	139.7s

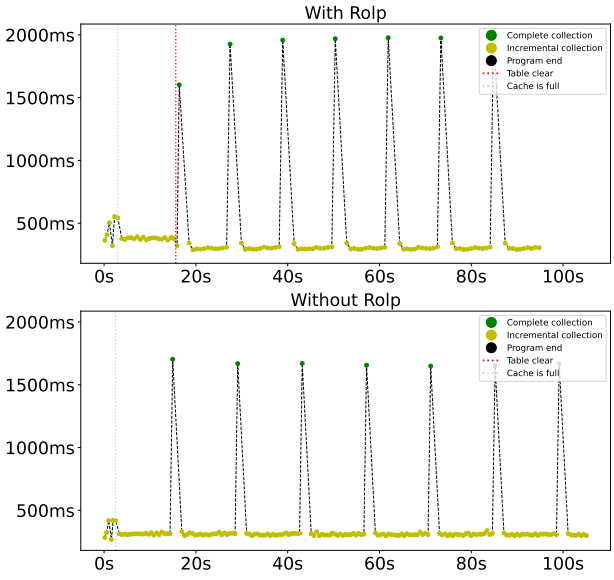
Table 6.4: Statistics for Vanilla for different *Circular Array* with only writing enabled.

Cache size	**Incremental**	**Complete**	**Collection time**	**Execution time**
10 000 000 (9%)	29	7	14.2s	111.70s
20 000 000 (19%)	32	8	20.0s	122.9s
40 000 000 (37%)	42	11	38.1s	176.7s
60 000 000 (56%)	56	17	72.4s	251.7s
80 000 000 (75%)	92	41	194.0s	456.5s

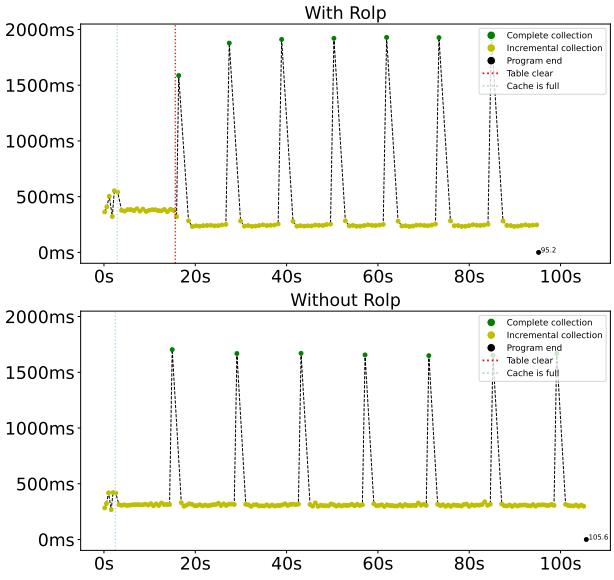
Table 6.5: Statistics for ROLP for different *Circular Array* runs with writing and reading enabled.

Cache size	**Incremental**	**Complete**	**Collection time**	**Execution time**
10 000 000 (9%)	124	6	35.6s	150.7s
20 000 000 (19%)	129	7	45.8s	174.5s
40 000 000 (37%)	143	9	71.1s	241.0s
60 000 000 (56%)	166	14	113.9s	358.0s
80 000 000 (75%)	156	28	183.0s	445.1s

Table 6.6: Statistics for vanilla for different *Circular Array* runs with writing and reading enabled.



(a) Collection times for hashmap.



(b) Collection times without FOT setup.

Figure 6.8: Collection times for *Circular Hashmap*.

6.2.4 Circular Hashmap

The options that we run the experiment with are:

- `-Xmx4g` for the maximum heap size. This results in a maximum heap of 4.3 GB and a young generation of 268 MB.
- 786 000 000 as total amount of allocations.
- 11 000 000 as key bound. This results in the hashmap occupying 1.45 GB of the heap (34%).
- `-H:FinalEpoch=24` to dictate the final profiling epoch to be 24.

We require 24 epochs (compared to the 8 of *Circular Array*), to capture the dynamic lifetime pattern of key-objects. All of the key-objects which are produced until we reach the *bound* of the hashmap are long-lived. When the hashmap is filled up, inserting key-values into the hashmap does not exchange the previous keys, due to the fact that they contain the same hashcode. Thus, the allocation site for the key objects is initially identified as long-lived. However, key-objects after the hashmap is filled up become short-lived. To capture this information of the key-objects, we have to increase the profiling phase up until the 24th epoch.

In this experiment, no reading from the hashmap is performed.

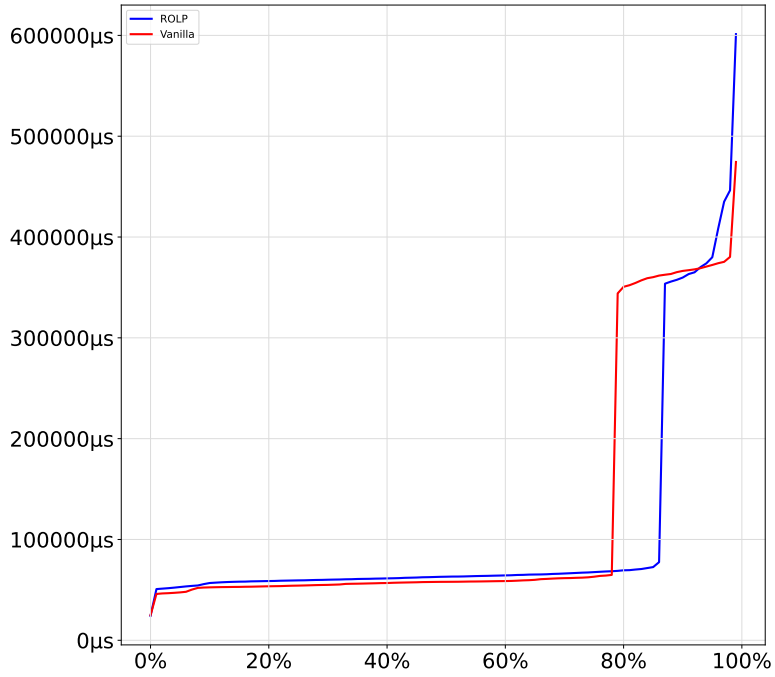


Figure 6.9: 0-99 percentile for *Circular Hashmap*. Each datapoint is the amount of time spent allocating 1 000 000 objects.

Collection Times

A significant difference in the collection times for *Circular Hashmap* is the prevalence of incremental collections for ROLP, as seen in Figure 6.8a. This is due to the fact that even though all of the values in the hashmap are identified as long-lived, all key objects generated after the hashmap has been filled up are only used to overwrite values. Since the key objects are already contained in the hashmap, they are not used beyond the overwriting of values and are thus identified as short-lived.

Even given the amount of incremental collections, ROLP has still a substantially amount lower of incremental collections. In this experiment, ROLP achieves 101 incremental collections compared to the 160 of vanilla. However, there seems to be an increase in individual complete collection time of 21% in Figure 6.8a, even if we do not consider FOT setup as seen in Figure 6.8b. We suspect this is the result of some unnecessary objects being scanned due to the FOT setup.

Overall, ROLP manages to reduce the total collection time by around 25% for the given run of *Circular Hashmap*. This results in a reduction of 10% of total execution time.

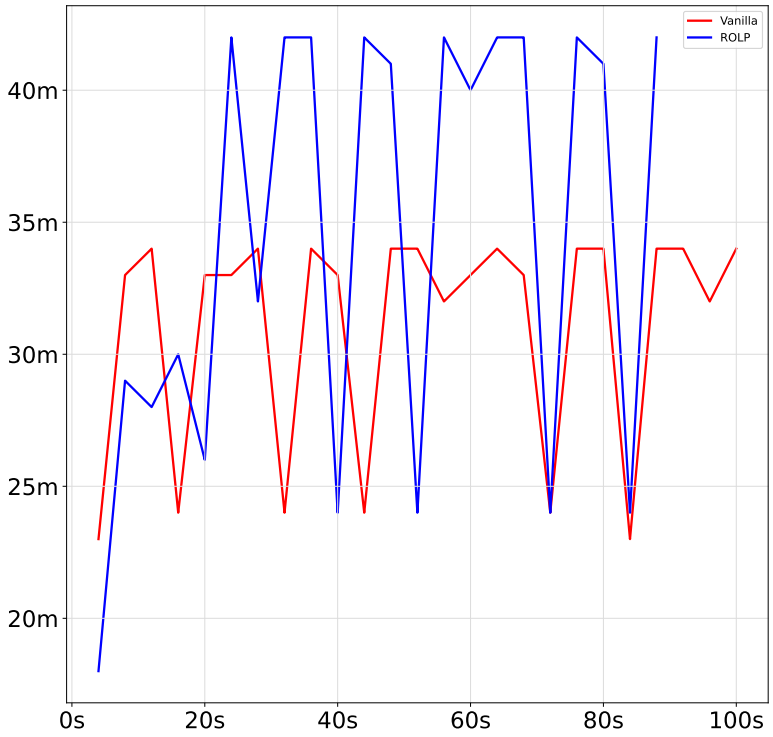


Figure 6.10: Throughput of *Circular Hashmao*. Each point contains objects allocated for intervals of 4 seconds.

Allocation Time Percentiles

From the allocation times seen in Figure 6.9, we can see that ROLP manages to delay the most lengthy allocations until the 88-90 percentiles. This is the result of the reduction of incremental collection. However, the highest percentiles have higher times with ROLP. This is the result of ROLP having longer complete collection times.

Throughput

In Figure 6.10, we can see that ROLP manages to achieve higher throughput. This is the result of less incremental collections impacting the allocations. The spikes which occur are a result of the lengthy incremental collection interruptions.

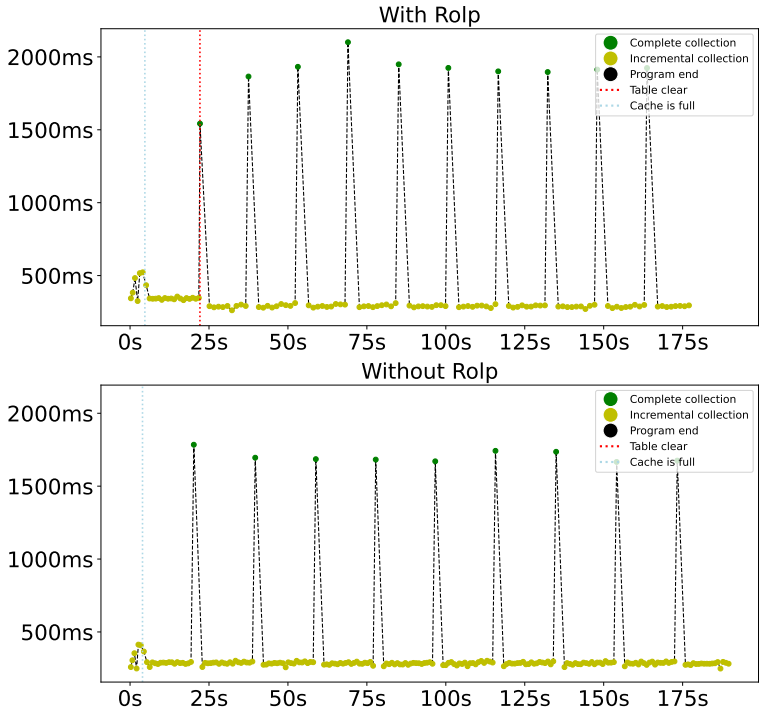
6.2.5 Circular Hashmap with Reading

The options that we run the experiment with are:

- `-Xmx4g` for the maximum heap size. This results in a maximum heap of 4.3 GB and a young generation of 268 MB.
- 786 000 000 as total amount of allocations.

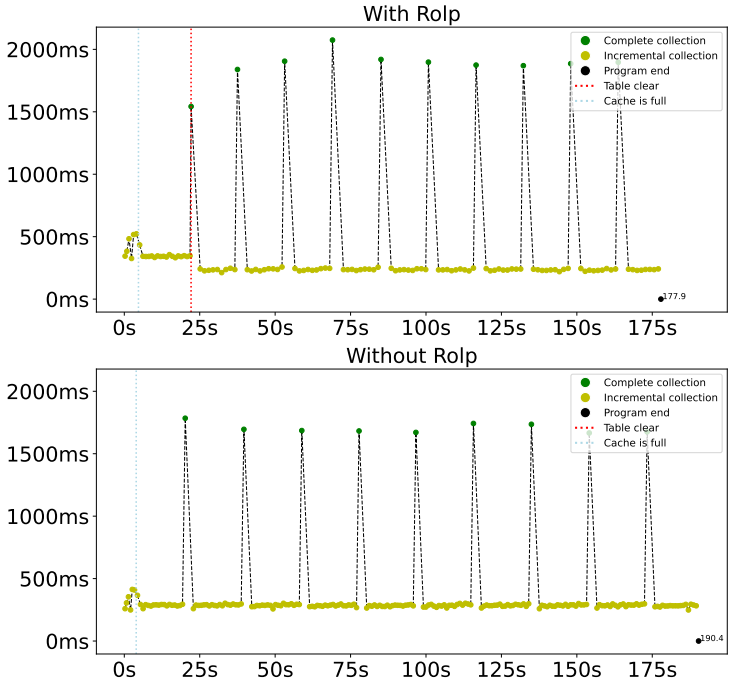
- 40 000 000 as cache size. This results in a fully-filled cache occupying 1.45 GB of the heap (34%).
- Read rate of 500 000 objects per second.
- `-H:FinalEpoch=24` to dictate the final profiling epoch to be 24.

We present the results on the following pages. The collection time results do not differ much compared to the *Circular Hashmap* with only writing. Application execution takes approximately 2x time to finish, both for Vanilla and ROLP. In addition there is an increase in incremental collections occurring due to short-lived objects produced by the reading. The amount of incremental collections equals 113 for ROLP and 193 for vanilla.



Total collection time 'Without Rolp' (ms): 70704.05692600002
Total collection time 'With Rolp' (ms): 53584.677217000004

Figure 6.11: Collection times for *Circular Hashmap* with reading.



Total collection time 'Without Rolp' (ms): 70704.05692600002
Total collection time 'With Rolp' (ms): 48544.38668

Figure 6.12: Collection times for *Circular Hashmap* without FOT setup (with reading).

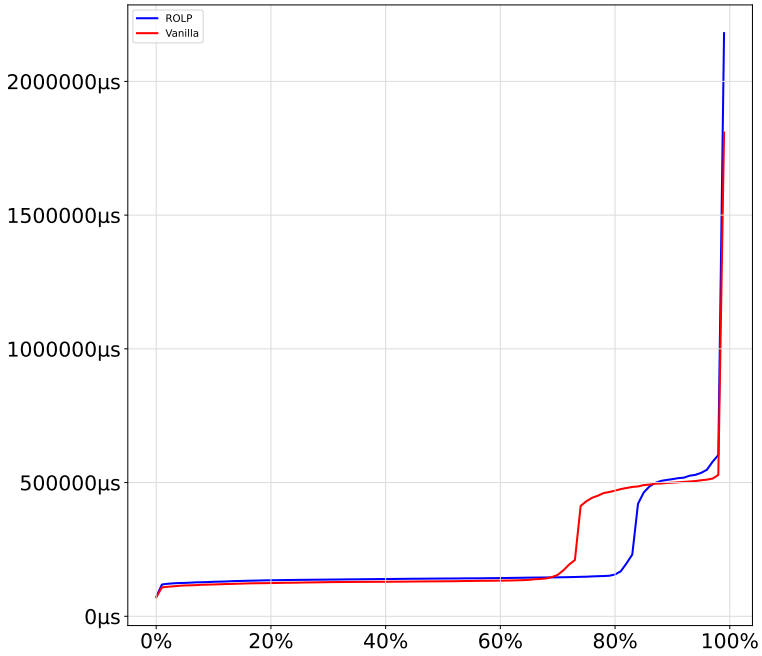


Figure 6.13: Allocation time percentiles for *Circular Hashmap* with reading.

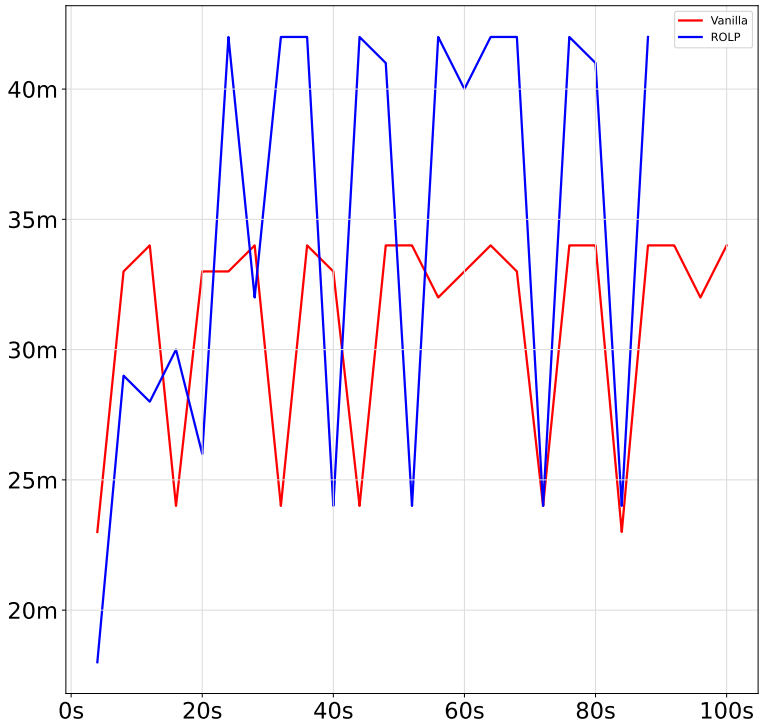


Figure 6.14: Throughput of *Circular Hashmap* with reading. The Y-axis is amount of objects allocated.

6.2.6 Discussing Results from Circular Hashmap

We provide condensed results from different *Circular Hashmap* runs in tables in the following pages. The columns represent the same values as presented in the previous tables for *Circular Array*, besides the value **key bound**, which depicts the bound value of the hashmap.

In Table 6.7 and Table 6.8 the parameters for total amount of allocations and heap size are the same as the ones defined for the *Circular Hashmap* run in Section 6.2.4. In the aforementioned tables we have strictly enabled writing without reading. In Table 6.9 and Table 6.10 we show results with enabled reading of 500 000 objects per second. The amount of reads is 4x lower than for *Circular Array* to avoid the lengthy execution times.

The results for amount of incremental collections occurring for ROLP and Vanilla are mostly similar as for *Circular Array*, with a significant decrease in incremental collections occurring for ROLP. A major difference is the amount of incremental collections occurring in ROLP, due to short-lived key objects. The collection time has also not as significant impact on total execution time as for *Circular Array*, due to the overhead of objects being allocated and the overhead from the ROLP collections.

However, much as the same way as for *Circular Array*, vanilla does not start outperforming ROLP until we reach a key bound value of around 22 000 000, which results in 68% of total heap consumption. ROLP is outperformed here due to the frequent and lengthy complete collections it performs, which occur due to a rapidly filled heap.

Key bound	**Incremental**	**Complete**	**Collection time**	**Execution time**
2 750 000 (8%)	102	7	22.5s	72.1s
5 500 000 (17%)	105	8	35.7s	88.6s
11 000 000 (34%)	102	10	52.1s	105.2s
16 500 000 (53%)	99	16	75.6	129.8s
22 000 000 (68%)	104	28	124.2s	178.6s

Table 6.7: Statistics for ROLP for different *Circular Hashmap* runs.

Key bound	**Incremental**	**Complete**	**Collection time**	**Execution time**
2 750 000 (8%)	184	4	38.9s	85.9s
5 500 000 (17%)	182	7	60.9	110.6s
11 000 000 (34%)	182	9	69.6s	120.2s
16 500 000 (53%)	178	14	86.1s	138.2s
22 000 000 (68%)	171	23	118.9.0s	171.4s

Table 6.8: Statistics for Vanilla for different *Circular Hashmap* runs.

Key bound	**Incremental**	**Complete**	**Collection time**	**Execution time**
2 750 000 (8%)	110	7	24.2s	124.7s
5 500 000 (17%)	113	8	37.7s	149.3s
11 000 000 (34%)	113	10	53.0s	175.5s
16 500 000 (53%)	114	16	76.7	209.9s
22 000 000 (68%)	106	28	126.4s	279.3s

Table 6.9: Statistics for ROLP for different *Circular Hashmap* runs with reading enabled.

Key bound	**Incremental**	**Complete**	**Collection time**	**Execution time**
2 750 000 (8%)	193	4	38.9s	142.3s
5 500 000 (17%)	193	7	60.9	178.5s
11 000 000 (34%)	194	9	69.6s	201.7s
16 500 000 (53%)	192	14	86.1s	226.7s
22 000 000 (68%)	185	25	118.9.0s	273.2s

Table 6.10: Statistics for Vanilla for different *Circular Hashmap* runs with reading enabled.

6.3 Summary

The results we have seen for the different experiments showcased that utilizing ROLP can reduce overall collection times significantly for allocation heavy applications. This results in higher overall throughput, not stagnated by incremental collections, which affect the Vanilla implementation due to prevalence of object copying. In the next chapter, the requirements we have set are addressed before presenting proposed future work. We finish of by presenting the conclusion.

Chapter 7

Conclusion

In this chapter we will start by discussing if we have met the requirements defined in Chapter 1. We will then present future work which might be possible to explore for the Native Image implementation of ROLP. We will finish of by discussing what we have explored throughout the thesis and our contributions.

7.1 Addressing the Requirements

In section 1.2 we outlined the requirements which our ROLP with Serial GC version aimed to fulfill. This is how our final implementation behaved according to them.

- Accurate pretenuring of objects in correlation to their allocation site and lifetime.

We have managed to implement a object profiler for GraalVM Native Image which allows the GC to pretenure objects according to their lifetime correctly with high reliability. This is apparent in the reduction of incremental collections, and the fact that necessary incremental collections take significantly less time. However, the accuracy of the pretenuring can depend on some factors, such as how many epochs the developer would like to pretenure for.

- Good throughput during application execution in comparison to a native image built without ROLP.

The evaluations we have shown in Chapter 6 have shown that throughput gain can be significant with ROLP, if there are objects throughout the application run which can be identified as long-lived. This is especially true if most objects tend to be long-lived, which eliminates the need for incremental collections significantly. There seems to be some overhead

when it comes to complete collection times for some applications (such as *Circular Hashmap*), but overall throughput is higher with ROLP.

- Good response time during application execution in comparison to a native image built without ROLP.

The evaluations we have shown in Chapter 6 has shown that for the experiments we have evaluated, GC pause times can be reduced significantly for applications in which certain objects tend to be long lived. This is done by eliminating the need for some incremental collections. In addition, the incremental collections which do occur avoid high amount of copying of objects.

7.2 Future Work

For future work the following can be explored:

- **Memory leaks** - One can utilize the lifetime information for each allocation site to detect if a memory leak has occurred. Since each allocation site contains statistics about how many promotions its objects have survived, one could detect if there is a unexpected number of objects either continuously surviving or being allocated. We have not explored this in our work since we have mainly focused on optimizing memory management decisions for the GC.
- **Integrating dynamic profiling and TSS** - These were implementation features which we were not capable to explore during this thesis. Enabling both of these features would allow increase the reliability of the pretenuring statistics and also enable long running applications with complex allocation patterns to perform better.
- **Evaluate with multiple survivor spaces** - Native Image allows one to specify the amount of survivor spaces the young generation should consist of. By default it only consists of a single eden space. By modifying `MaxSurvivorSpaces` option, one can enable multiple survivor spaces between the young generation and the old one. It would be interesting to see the results of pretenuring objects directly into those spaces, and if it provides any more improvements towards GC pause times.
- **Evaluate with a realistic benchmark** - The experiments we have covered have been setup to mimic allocation heavy workloads. Unfortunately, due to the time limitations of a master thesis, we did have the capability to evaluate with a more realistic service-based workload. This is something that could be explored in conjunction with a microservice framework, to see if ROLP could perform well in a more realistic environment.

- **Integration with Truffle** - Integrating ROLP with the Truffle Language Implementation Framework would allow us to enable object pretenuring for a multitude of programming languages. There already exists implementations of the Python and Ruby languages on GraalVM based on the latest standards, in addition to several other popular languages. The amount of work to enable ROLP for these languages should not be a significant, as most of the ground work is done.

7.3 Conclusion

In this thesis, we presented an object profiler known as ROLP for the GraalVM Native Image which enabled object pretenuring for the native GC. Through the utilization of ROLP, we showcased results with significant reductions in GC pause times and increased throughput for a number of applications built using the utility.

Enabling the profiler known as ROLP does not require significant developer effort, working out-of-the-box for most application in which object promotion is prevalent. With minor tweaking of the build and runtime options, it can perform well for most applications.

We believe that ROLP is an important asset in object-oriented runtimes, since they often deal with generational garbage collectors. The downside of generational collection algorithms is that minor collections can occur frequently, promoting a large number of objects which in turn impends on the overall performance. By identifying often promoted objects through the usage of ROLP, one can prevent a majority of collections by allocating these objects directly in the survivor space.

Bibliography

[1] Bowen Alpern, Steve Augart, Stephen Blackburn, Maria Butrico, Anthony Cocchi, Perry Cheng, Julian Dolby, Stephen Fink, David Grove, Michael Hind, Kathryn McKinley, Mark Mergen, Eliot Moss, Ton Ngo, Vivek Sarkar and Martin Trapp. 'The Jikes Research Virtual Machine project: Building an open-source research community'. In: *IBM Systems Journal* 44 (Jan. 2005), pp. 399–418. DOI: 10.1147/sj.442.0399.

[2] Lars Bak. 'Google Chrome's Need for Speed'. In: *Chromium Blog* (2nd Sept. 2008). URL: https://blog.chromium.org/2008/09/google-chromes-need-for-speed_02.html (visited on 11/04/2021).

[3] Edd Barrett, Carl Friedrich Bolz-Tereick, Rebecca Killick, Sarah Mount and Laurence Tratt. 'Virtual Machine Warmup Blows Hot and Cold'. In: *Proc. ACM Program. Lang.* 1.OOPSLA (Oct. 2017). DOI: 10.1145/3133876. URL: https://doi.org/10.1145/3133876.

[4] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage and B. Wiedermann. 'The DaCapo Benchmarks: Java Benchmarking Development and Analysis'. In: *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications*. Portland, OR, USA: ACM Press, Oct. 2006, pp. 169–190. DOI: http://doi.acm.org/10.1145/1167473.1167488.

[5] Stephen M Blackburn, Sharad Singhai, Matthew Hertz, Kathryn S McKinely and J Eliot B Moss. 'Pretenuring for java'. In: *ACM SIGPLAN Notices* 36.11 (2001), pp. 342–352.

[6] Rodrigo Bruno and Paulo Ferreira. 'A study on garbage collection algorithms for big data environments'. In: *ACM Computing Surveys (CSUR)* 51.1 (2018), pp. 1–35.

[7] Rodrigo Bruno and Paulo Ferreira. 'POLM2: automatic profiling for object lifetime-aware memory management for hotspot big data applications'. In: *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference*. 2017, pp. 147–160.

[8] Rodrigo Bruno, Luıs Picciochi Oliveira and Paulo Ferreira. 'NG2C: Pretenuring Garbage Collection with Dynamic Generations for HotSpot Big Data Applications'. In: *Proceedings of the 2017 ACM SIGPLAN International Symposium on Memory Management*. ISMM 2017. Barcelona, Spain: Association for Computing Machinery, 2017, pp. 2–13. ISBN: 9781450350440. DOI: 10.1145/3092255.3092272. URL: https://doi.org/10.1145/3092255.3092272.

[9] Rodrigo Bruno, Duarte Patricio, José Simão, Luis Veiga and Paulo Ferreira. 'Runtime object lifetime profiler for latency sensitive big data applications'. In: *Proceedings of the Fourteenth EuroSys Conference 2019*. 2019, pp. 1–16.

[10] Chris J Cheney. 'A nonrecursive list compacting algorithm'. In: *Communications of the ACM* 13.11 (1970), pp. 677–678.

[11] Perry Cheng, Robert Harper and Peter Lee. 'Generational stack collection and profile-driven pretenuring'. In: *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*. 1998, pp. 162–173.

[12] Daniel Clifford, Hannes Payer, Michael Stanton and Ben L Titzer. 'Memento mori: dynamic allocation-site-based optimizations'. In: *ACM SIGPLAN Notices* 50.11 (2015), pp. 105–117.

[13] Sébastien Deleuze. 'Spring Native for GraalVM 0.8.3 available now'. In: *spring.io* (23rd Nov. 2020). URL: https://spring.io/blog/2020/11/23/spring-native-for-graalvm-0-8-3-available-now (visited on 11/04/2021).

[14] David Detlefs, Christine Flood, Steve Heller and Tony Printezis. 'Garbage-first garbage collection'. In: *Proceedings of the 4th international symposium on Memory management*. 2004, pp. 37–48.

[15] Gilles Duboscq, Thomas Würthinger, Lukas Stadler, Christian Wimmer, Doug Simon and Hanspeter Mössenböck. 'An intermediate representation for speculative optimizations in a dynamic compiler'. In: *Proceedings of the 7th ACM workshop on Virtual machines and intermediate languages*. 2013, pp. 1–10.

[16] Roy Fielding, Jim Gettys, Jeffrey Mogul, Henrik Frystyk, Larry Masinter, Paul Leach and Tim Berners-Lee. 'Hypertext transfer protocol–HTTP/1.1'. In: (1999).

[17] GraalVM. *mx*. https://github.com/graalvm/mx. 2019.

[18] Wilhelm Hasselbring. 'Microservices for scalability: Keynote talk abstract'. In: *Proceedings of the 7th ACM/SPEC on International Conference on Performance Engineering*. 2016, pp. 133–134.

[19] P. Jamshidi, C. Pahl, N. C. Mendonça, J. Lewis and S. Tilkov. 'Microservices: The Journey So Far and Challenges Ahead'. In: *IEEE Software* 35.3 (2018), pp. 24–35.

[20] Richard E Jones. *Garbage collection: algorithms for automatic dynamic memory management*. John Wiley and Sons, 1996.

[21] Ian Joyner. 'A Critique of C++ and Programming and Language Trends of the 1990s'. In: *Available by any good Internet search* (1996).

[22] David Lion, Adrian Chiu, Hailong Sun, Xin Zhuang, Nikola Grcevski and Ding Yuan. 'Don't get caught in the cold, warm-up your {JVM}: Understand and eliminate {JVM} warm-up overhead in data-parallel systems'. In: *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*. 2016, pp. 383–400.

[23] Iván López and Sergio del Amo. *Creating your first Micronaut Graal application*. 2021. URL: https://guides.micronaut.io/micronaut-creating-first-graal-app/guide/index.html.

[24] Sebastien Marion, Richard Jones and Chris Ryder. 'Decrypting the Java gene pool'. In: *Proceedings of the 6th international symposium on Memory management*. 2007, pp. 67–78.

[25] John McCarthy. 'Recursive functions of symbolic expressions and their computation by machine, part I'. In: *Communications of the ACM* 3.4 (1960), pp. 184–195.

[26] *Memory Management at Image Run Time*. URL: https://www.graalvm.org/reference-manual/native-image/MemoryManagement/.

[27] Fabio Niephaus, Tim Felgentreff and Robert Hirschfeld. 'Towards polyglot adapters for the graalvm'. In: *Proceedings of the conference companion of the 3rd international conference on art, science, and engineering of programming*. 2019, pp. 1–3.

[28] Michael Redlich. *Helidon Supports GraalVM for Native Executable Applications*. July 2019. URL: https://www.infoq.com/news/2019/07/helidon-supports-graalvm/.

[29] Leonard Richardson and Sam Ruby. *RESTful web services*. " O'Reilly Media, Inc.", 2008.

[30] Oleg Šelajev. 'Lightweight cloud-native Java applications'. In: *Medium* (28th May 2019). URL: https://medium.com/graalvm/lightweight-cloud-native-java-applications-35d56bc45673 (visited on 06/05/2021).

[31] M Šipek, B Mihaljević and A Radovan. 'Exploring Aspects of Polyglot High-Performance Virtual Machine GraalVM'. In: *2019 42nd International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*. IEEE. 2019, pp. 1671–1676.

[32] M Šipek, D Muharemagić, B Mihaljević and A Radovan. 'Enhancing Performance of Cloud-based Software Applications with GraalVM and Quarkus'. In: *2020 43rd International Convention on Information, Communication and Electronic Technology (MIPRO)*. IEEE, pp. 1746–1751.

[33] Davide Taibi, Valentina Lenarduzzi and Claus Pahl. 'Processes, motivations, and issues for migrating to microservices architectures: An empirical investigation'. In: *IEEE Cloud Computing* 4.5 (2017), pp. 22–32.

[34] David Tarditi, Greg Morrisett, Perry Cheng, Chris Stone, Robert Harper and Peter Lee. 'TIL: A type-directed optimizing compiler for ML'. In: *ACM Sigplan Notices* 31.5 (1996), pp. 181–192.

[35] Johannes Thönes. 'Microservices'. In: *IEEE software* 32.1 (2015), pp. 116–116.

[36] Jaroslav Tulach. 'Improving performance of GraalVM native images with profile-guided optimizations'. In: *Medium* (28th May 2019). URL: https://medium.com/graalvm/improving-performance-of-graalvm-native-images-with-profile-guided-optimizations-9c431a834edb (visited on 07/05/2021).

[37] David Ungar. 'Generation scavenging: A non-disruptive high performance storage reclamation algorithm'. In: *ACM Sigplan notices* 19.5 (1984), pp. 157–167.

[38] Mario Villamizar, Oscar Garces, Lina Ochoa, Harold Castro, Lorena Salamanca, Mauricio Verano, Rubby Casallas, Santiago Gil, Carlos Valencia, Angee Zambrano et al. 'Infrastructure cost comparison of running web applications in the cloud using AWS lambda and monolithic and microservice architectures'. In: *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*. IEEE. 2016, pp. 179–182.

[39] Christian Wimmer, Codrut Stancu, Peter Hofer, Vojin Jovanovic, Paul Wögerer, Peter B. Kessler, Oleg Pliss and Thomas Würthinger. '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.

[40] Thomas Würthinger. 'Visualization of Program Dependence Graphs'. MA thesis. Altenberger Straße 69: Johannes Kepler University Linz, 2007.

[41] Thomas Würthinger, Christian Wimmer, Christian Humer, Andreas Wöß, Lukas Stadler, Chris Seaton, Gilles Duboscq, Doug Simon and Matthias Grimmer. 'Practical partial evaluation for high-performance dynamic language runtimes'. In: *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2017, pp. 662–676.