UPPSALA
UNIVERSITET

UPTEC IT 20019
Examensarbete 30 hp
Juni 2020

Moving Garbage Collection with Low-Variation Memory Overhead and Deterministic Concurrent Relocation

Jonas Norlinder

Institutionen för informationsteknologi
Department of Information Technology

UPPSALA
UNIVERSITET

Teknisk- naturvetenskaplig fakultet
UTH-enheten

Besöksadress:
Ångströmlaboratoriet
Lägerhyddsvägen 1
Hus 4, Plan 0

Postadress:
Box 536
751 21 Uppsala

Telefon:
018 – 471 30 03

Telefax:
018 – 471 30 00

Hemsida:
http://www.teknat.uu.se/student

Abstract

Moving Garbage Collection with Low-Variation Memory Overhead and Deterministic Concurrent Relocation

Jonas Norlinder

A parallel and concurrent garbage collector offers low latency spikes. A common approach in such collectors is to move objects around in memory without stopping the application. This imposes additional overhead on an application in the form of tracking objects' movements, so that all pointers to them, can eventually be updated to the new locations. Typical ways of storing this information suffer from pathological cases where the size of this "forwarding information" can theoretically become as big as the heap itself. If we dimension the application for the pathological case this would be a waste of resources, since the memory usage is usually significantly less. This makes it hard to determine an application's memory requirements.

In this thesis, we propose a new design that trades memory for CPU, with a maximum memory overhead of less than 3.2% memory overhead. To evaluate the impact of this trade-off, measurements on application execution time was performed using the benchmarks from the DaCapo suite and SPECjbb2015. For 6 configurations in DaCapo a statistically significant negative effect on execution time in the range of 1-3% was found for the new design. For 10 configurations in DaCapo no change in execution times was shown in statistically significant data and two configurations in DaCapo showed a statistically significant shorter execution times for the new design on 6% and 22%, respectively. In SPECjbb2015, both max-jOPS and critical-jOPS has, for the new design, a statistically significant performance regression of ~2%. This suggests that for applications were stable and predictable memory footprint is important, this approach could be something to consider.

Handledare: Erik Österlund, Per Lidén
Ämnesgranskare: Tobias Wrigstad
Examinator: Lars-Åke Nordén
UPTEC IT 20019
Tryckt av: Reprocentralen ITC

Contents

List of Figures

List of Tables

Chapter 1

Introduction

A garbage collector collects memory garbage. Memory is considered to be garbage when it is no longer reachable from a normal user thread. With applications that leverage modern multicore architectures comes a need for *concurrent garbage collection* to avoid latency spikes. A garbage collector is concurrent if it is able to collect garbage concurrently with the execution of normal user threads [1].

To allow fast allocation in garbage collected environments, a common approach is to use bump pointer allocation [1]. Bump pointer allocation uses a pointer to the first available byte in memory that is monotonically increased as the application continues to allocate objects. While this scheme allows fast allocation it also comes with a caveat that the free memory must be kept continuous. To keep the free memory continuous, many garbage collectors move objects around in memory to compact them, thereby avoiding fragmentation. This is typically handled in a process where all live objects are moved off of a page which is then free'd. This permits moving all living objects off of a page in $\mathcal{O}(live)$ objects, which is typically a small number (relatively speaking) due to the weak generational hypothesis [1].

The moving of objects introduces the task of updating all incoming pointers to moved objects to point to their new location. This typically involves a mapping from old addresses to new addressen, which is commonly known as forwarding information. Forwarding information is commonly stored in the space of a moved object (reusing that space) or in an auxiliary data structure. In concurrent garbage collectors, the mutator – *i.e.*, an application's threads in garbage collector parlance [1] – may access objects at any time. This requires any forwarding information to always be up-to-date.

Floating garbage is garbage that for some reason was not collected during a garbage collection cycle [1]. When moving an object, its previous location can be used for storing auxiliary data. At face value, reusing the memory of a moved object to store its forwarding pointer is space-efficient. What may be less apparent is that it may lead to increased amounts of floating garbage and additional latency before some space can be reclaimed. This is because the old copy must be retained until we are sure that all pointers to the old object have been updated to the new address and this leads. If objects are not free'd individually, but rather as the side-effect of moving all objects on a memory page (some contiguous space), the effect can be exacerbated.

Memory pressure is correlated with the allocation rate and implicitly with the reclamation rate. If the allocation rate is high and reclaiming rate is low, the memory pressure is high. If the allocating rate is high and the reclaiming rate is high the

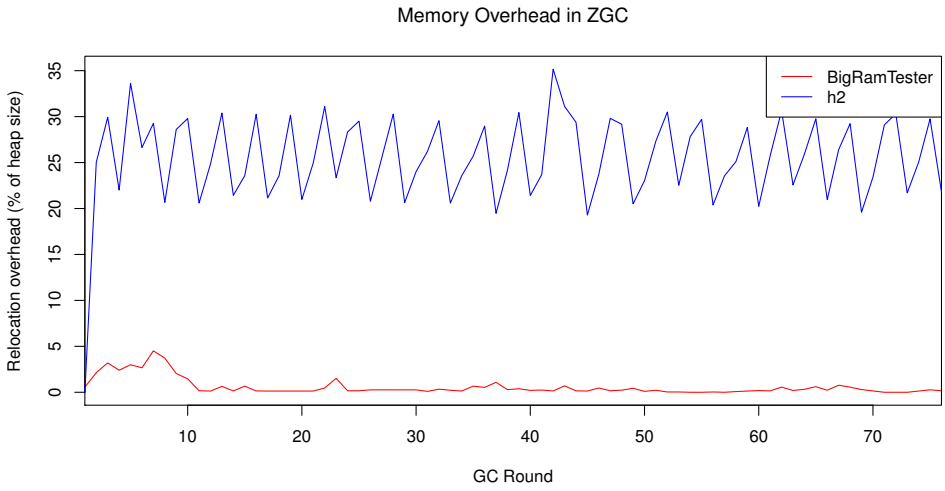


Figure 1.1: Memory overhead due to forwarding information in the Z garbage collector as observed in two benchmarks, BigRamTester and h2, with a peak memory overhead of 35% for BigRamTester and 4.5% for h2.

memory pressure *can* be low. Low allocation rate would always imply low memory pressure. Since one floating garbage object can prevent a whole page from being recycled, memory pressure increases.

Floating garbage is reduced when using a separate data structure to hold the forwarding information (typically referred to as a forwarding table) as no information in the moved objects (or the entire memory page) is needed to remap incoming pointers. Naturally, the forwarding table incurs additional overhead in the form of one entry per relocated object.

Knowing the total memory usage of an application (application and auxiliary data needed by the programming language implementation to run it), is needed to properly size memory of the system running the application.

Some languages offer option to specify an application's maximum (and minimum) heap size. If this value is too big, it might impact performance in various ways, *e.g.,* due to bad data locality or unnecessarily long garbage collection pauses. If the value is too low, the application might crash due to an out-of-memory error. If the memory requirements overshoot available DRAM, performance might drop considerably due to paging and swapping memory on to disk.

Understanding the total memory requirements for an application is not easy as it depends on many factors which cannot be known without running the application with a representative load. The average case scenario can often be "guesstimated" – but in the worst case scenario, it might be significantly higher in than the average case. This is exemplified by Figure 1.1, showing the memory overhead of the auxiliary data structure holding forwarding information in OpenJDK using the Z Garbage Collector (ZGC) [2]. The data in the figure shows two benchmarks, BigRamTester and h2, with a peak memory overhead of 35% for BigRamTester and 4.5% for h2. The upper limit of the memory overhead for storing the forwarding information is currently 100% in ZGC and this overhead depends on the life cycle of the objects, consequently it can be hard to reason about the lower and upper limits

of the total memory usage of an application. ZGC is one of the garbage collectors available in OpenJDK which is an open-source implementation of Java. Properly understanding the memory requirements for an application is important to avoid out-of-memory crashes (by setting the maximum heap size too low) or unnecessary slow-downs and wasting DRAM (by setting the maximum heap size too high).

1.1 Purpose and Goals

The purpose of this work is to simplify the the task of determining the memory requirements for applications running on a Java Virtual Machine (JVM). The specific goal for this thesis is to design and implement an alternative approach to object forwarding in the Z garbage collector that, without slowing down the JVM considerably, in contrast to what Fig. 1.1 showed, has a low and predictable memory overhead. While we use ZGC as a context vehicle to evaluate our design, our approach is not specific to ZGC. It could, at least in theory, be applicable also to *e.g.,* Shenandoah and ART's concurrent collector, to reduce the memory overheads inherent in the design of their storage of forwarding information.

1.2 Thesis Outline

Chapter 2 argues why garbage collection is important in modern software development, and follows-up with a coverage of the parts necessary to understand our proposed solution. Chapter 3 gives a high-level description of the solution. Measuring performance in a non-deterministic environment like the JVM is a non-trivial task. Chapter 4 covers the approach taken. Chapter 5 shows the results of the evaluation. Finally, Chapter 6 concludes.

Chapter 2

Garbage Collection

Garbage collection refers to an automated process through which the run-time system identifies objects which can be safely deallocated to free memory resources [1]. In this chapter we, begin with introducing the need for garbage collection. Then we describe the commonly used generational garbage collection algorithm to give the reader a sense of how ZGC fits into the landscape of state-of-the art garbage collectors. We also briefly discuss Android Runtime to show that this thesis have a broader applicability than just within the OpenJDK project. Finally a short description of the OpenJDK project is presented and a detailed description of the garbage collector that the presented design is implemented on top of, ZGC is given.

A garbage-collected program consists of two semi-independent components: one or more mutator threads and one or more collector threads [3]. The mutator threads are assumed to have a finite set of roots, which are a special case of references. We denote something as a root if the reference is accessible to the mutator without traversing any objects. If there is a path to an object from a root, that object is said to be *live*. If there is no path to an object from a root in the system, then the object is unreachable and is considered garbage and may be safely collected. The number of living objects in the heap are referred to as the *live-set*.

The full set of activities of garbage collection are typically referred to as a *garbage collection cycle* [1] Some collector designs are stop-the-world designs, which means that the mutator threads are paused during (at least parts of) the garbage collection cycle. Depending on how a collector handles deallocation, a garbage collection cycle may or may not collect all garbage objects. Garbage that survives a GC cycle is referred to as *floating garbage*. In the case of concurrent garbage collectors, stop-the-world events may or may not happen but are designed to be as brief as possible, to avoid pausing the mutators. This typically means that garbage collecting activities run concurrently with the mutator. The duration of a stop-the-world pause is referred to as *pause time*.

Some programming languages support destructors or finalizers which are methods in an object triggered right before the object is garbage collected. In some languages, such methods may “resurrect” an object by creating a pointer to the object about to be deleted in another live object, or in a root [1]. This will not be discussed further in this thesis, since the forwarding information that this thesis is about, is orthogonal with regards to that.

Bump pointer allocation is a common allocation strategy in garbage collected languages as it offers good performance – both in terms of fast allocation and good locality of reference. In bump pointer allocation, an allocation buffer has a pointer

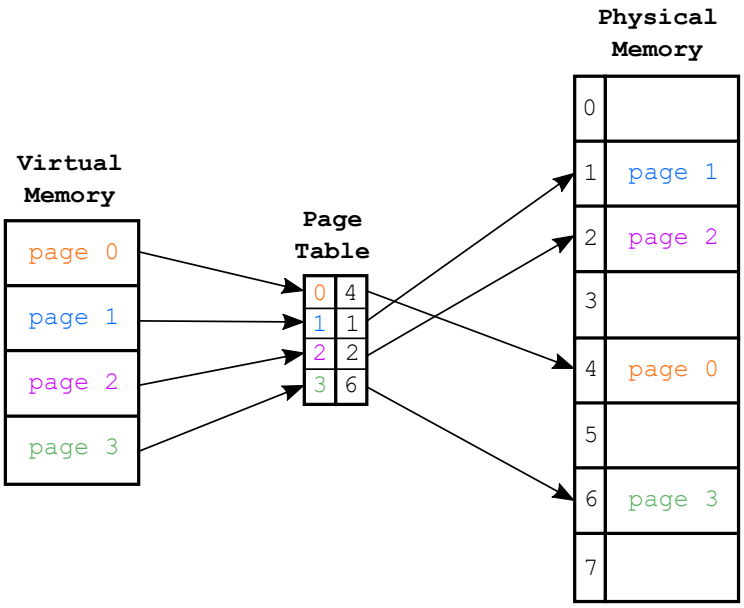


Figure 2.1: Virtual memory is mapped to physical memory using a page table. An object that spans the virtual pages 0–1 may be logically in a continuous address space, but physically placed in a non-continuous address space. The arrows indicates where the memory is mapped to.

p to the start of the free memory, and allocating n bytes simply means returning the current value of p and then moving p by n bytes for the next allocation. This strategy requires the free memory to be contiguous and thus, is prone to fragmentation. The fragmentation introduced by bump pointer allocation is typically reduced by paging. Paging is a common memory-management strategy that provides a continuous address space for applications [4]. It maps varying sizes of chunks of virtual memory that are logically continuous, to a physical location which may be non-continuous, using a page table. Since paging utilizes a page table, these chunks can be placed anywhere in the physical memory, as depicted in Figure 2.1. Since the virtual chunks can be physically placed anywhere this may lead to better utilization of memory, *i.e.,* less fragmentation of the physical memory. While fragmentation between pages, referred to as external fragmentation, is minimized using paging, internal fragmentation within a page is still a problem if used with a bump pointer allocation strategy. So garbage collectors that uses paging and bump pointer allocation usually design the collection cycle to also decrease internal fragmentation by moving all living objects from one or more pages to new pages. This means that the new pages will be populated with continuously live objects, *i.e.,* no fragmentation and the old pages can be recycled without having to explicitly free all their objects (modulo destructors). This type of garbage collector is known as a moving garbage collector, since it moves objects around as part of its collection cycle.

2.1 The Need for Garbage Collection

Most non-trivial applications make use of dynamic memory allocation. Dynamic memory allocation allows objects to be allocated and deallocated during run-time,

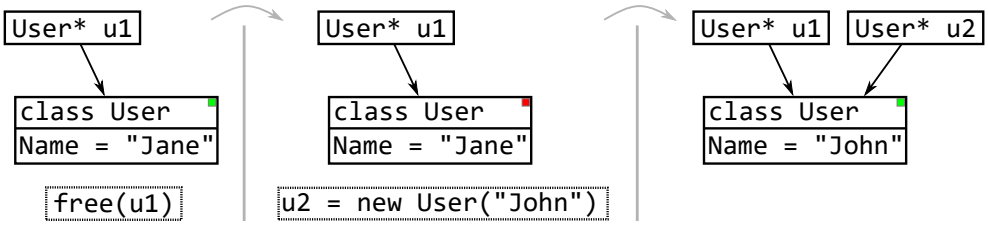


Figure 2.2: The dangers of dangling pointers. The object pointed to from `u1` is prematurely free'd and its space reused by a new `User` at the same starting address. Green indicates that the memory is claimed and red indicates that the memory is unclaimed. This kind of bug may be very hard to detect since it may not cause a crash.

and sizes of objects, as well as their location in memory, to be determined at runtime rather than at compile-time. Objects that are dynamically allocated are typically stored on the heap and not the stack which decouples their life-time from the stack frame of the allocating function [1]. For this thesis, all dynamic object allocation takes place on the heap.

2.1.1 The Deallocation Dilemma—When to Deallocate?

The more complicated and dynamic the allocation scheme of a program, the harder it is to determine when an object can be safely deallocated to free up space for new objects. Determining whether an object is reachable from the application (live)—or unreachable, is in the general case a global problem that cannot be solved by local reasoning. This is brought on by the fact that in most mainstream languages, pointers which are aliased are indistinguishable from pointers which are not. Thus, one cannot typically deduce from local reasoning whether the object pointed to by the variable `x` is reachable by other places in the code, and whether overwriting the contents of `x` overwrites the last reference to an object which should therefore be deallocated.

Timely deallocation of objects is an important consideration for a program. Deallocating an object too early gives rise to a dangling pointer whose subsequent accesses may read inconsistent data (for example if the data has been partially overwritten by new objects, see Figure 2.2 for details). This may corrupt the program or produce incorrect results and is a common attack vector for programs. If objects are allocated too late, a program might instead run out of memory, or suffer performance regression due to swapping. In some languages, additional complexity arises from the lack of support for multiple deallocation of the same object, *e.g.,* in C[1] and C++[2] attempting to deallocate an already deallocated object is undefined behaviour.

[1] `https://en.cppreference.com/w/c/memory/free`

[2] `https://en.cppreference.com/w/cpp/memory/c/free`

2.1.2 Delegating Deallocation to Garbage Collectors

Garbage collectors address the deallocation dilemma by using more or less automated techniques that discover and reclaim objects that cannot be reached by the mutator [1]. A simple garbage collection algorithm that over-approximates liveness uses a counter for all incoming references to an object and only deallocates objects

whose counter reaches zero. The reason why this is an over-approximation is because the existence of a reference to an object does not necessarily mean that a mutator will access the object at some point in the future. Another approach, which is also an approximation for the same reason, is to equate liveness with reachability, which can be decided by travsersing the pointer structures in the heap from the roots. Garbage collectors that traverse the heap are typically called *tracing* collectors.

Garbage collectors must be conservative in their estimation of liveness to avoid premature deallocation. The cost of over-approximating live objects gives rise to floating garbage, *i.e.,* objects that are kept alive after a garbage collection cycle despite they will not be used anymore. This may result in out-of-memory errors or triggering collections which will significantly slow down the application in order to find garbage. Some floating garbage is often accepted, since prematurely deallocate objects causes severe issues and as described in Section 2.1.1, this may lead to a non-functional program, or because it is more efficient.

In a software engineering context, the major benefit of using garbage collection is that it allows developers to write modules that have high cohesion and low coupling. Explicit memory management forces the modules to consider other modules [1]. From an economic perspective, it may desirable to avoid explicit memory management in order to increase productivity and lower development costs [1]. In a 1985 study by Rovner [5], it was approximated that the development team for Xerox's Mesa system spent 40% of their time implementing correct explicit memory management.

When using bump pointer allocation, fragmented memory is problematic. Consider the left hand side in Figure 2.3 and that we want to allocate memory for an object that would fit into one row. While there is space available between A–D and D–F, we cannot access this address space because bump pointer allocation only allows for allocating monotonically increasingly addresses. Relocating memory to eliminate fragmentation is called compaction and is depicted in Figure 2.3. Many collectors free memory as part of compaction. When all live objects have been moved off of a page and copied into continuous memory on some other page, all objects on the first page can be implicitly free'd in constant time by recycling the entire page [1]. This enables reclaimaing in order of live objects as opposed to order of garbage objects, which is typically faster given the weak generational hypothesis (see below) [1].

2.2 Generational Garbage Collection

A source of performance regression, especially in tracing collectors, is the repeated processing of long-lived objects, since this incurs a cost without—typically—any benefits. A generational garbage collector attempts to lower this cost by mapping objects on the heap into different generations depending on their age. For simplicity, consider two generations which we will call young and old (several generations are possible). The idea is to process the young generation often and the old generation more infrequently. This is beneficial because tracing is $\mathcal{O}$(live objects) and most young objects will be garbage, meaning great return on investment with regards to reclamation and minimal repeated work on old objects. That objects tend to die young is also known as the *weak generational hypothesis* [1]. As the weak generational hypothesis is assumed to hold for most programs, a generational garbage collector is (in some shape) a common GC design pattern.

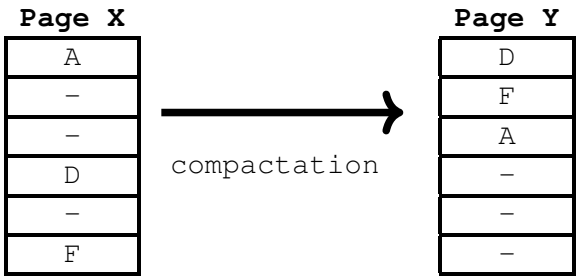


Figure 2.3: Memory defragmentation. The page to the left is fragmented. Through compaction the fragmentation is removed in the page to the right. Most schemes also allow the objects to be relocated in a different order and even to different pages.

Traditionally, generational garbage collectors divide the heap into different spaces, one part for each generation. Objects are born in the youngest generation [1]. Objects that survive n amount of garbage collector cycles may get promoted. Promoting an object to another generation involves copying it from one heap space into the other. Typically a *minor* collection collects only young generations and is performed often, while a *major* collection collects all generations and is performed more infrequently, thereby avoiding trying to collect long-lived objects.

To avoid tracing the entire heap (which would defeat the purpose of dividing the heap into multiple spaces) special measures must be taken when performing a minor collection [1]. Pointers from older generations to objects in the younger generation are usually referred to as intra-generational pointers. If a minor collection is performed and intra-generational pointers are not considered, then objects may get prematurely free'd. Generational garbage collectors store all intra-generational pointers in what is usually referred to as a *remembered set*. A remembered set is some data structure that has the invariant that it must contain all intra-generational pointers for that particular generation.

As depicted in Figure 2.4, a remembered set is needed such that object `D` is not collected during a minor collection. To maintain the invariant that the remembered set always contain all intra-generational pointers, the mutator must update the remembered set as such pointers are created or destroyed. This is typically achieved using write barriers. A write barrier is a special piece of code which will be executed when writing to memory, which will update the remembered set as needed.

2.3 Garbage Collection in the Android Runtime

Android is the leading operating system on smart phones, with a market share of 72% (March, 2020) [6]. Android uses a Java runtime and while other languages are supported, applications for Android are primarily written in Java. Some parts of Android uses code from the open-source implementation of Java, OpenJDK. The Android runtime (ART) provides its own garbage collector (not shared with OpenJDK), which is a concurrent moving collector [7] since version 8.0 (released 2017), and a concurrent moving generational collector since version 10.0 (released 2020) [8]. In

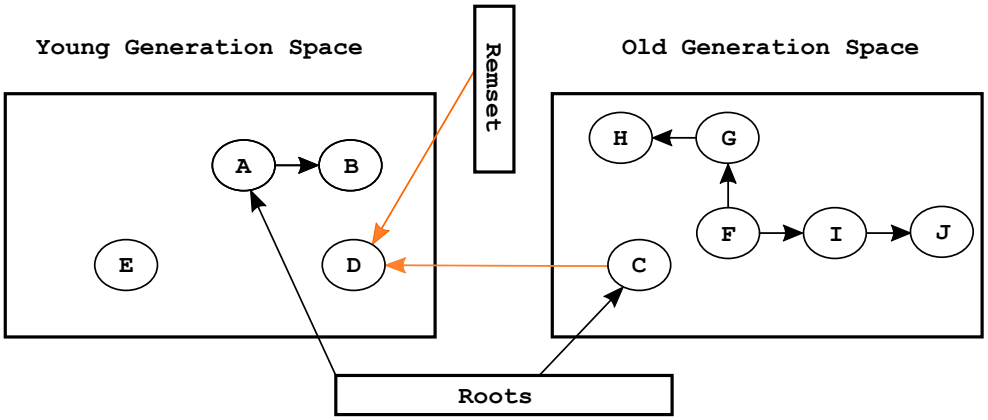


Figure 2.4: A remembered set voids the need to traverse the entire heap to avoid prematurely freeing objects. Doing a minor garbage collection on the young generation space without considering any pointers from the old generation space could lead to prematurely freeing an object. The circles represents objects and the arrows represents a pointer to the objects.

a concurrent moving collector, objects are relocated as the mutator is running. This creates a need for storing forwarding information in a compact way. To accomplish this, they use the old space in moved objects to store the new location.

2.4 OpenJDK

OpenJDK is an open-source implementation of Java. It provides a Java Development Kit (JDK) which includes the Java Runtime Environment (JRE) along with tools to compile and debug Java applications. The JRE consists of the Java Virtual Machine (JVM) and libraries [9].

The JDK is used daily by millions of people and thousands of businesses [10]. It is a large and complex software engineering project. There are currently 15 497 289 [3] number of lines of code in the entire OpenJDK. Making changes can, therefore, be a delicate process and the bar of entry for a new developer of OpenJDK is high. Some parts of the code are old and can be hard to read. One example is optimizations introduced 20 years ago which are not needed today, like efficient usage of each bit.

OpenJDK offers a wide range of garbage collectors, which can easily be changed by an input parameter when starting the JVM. There is currently a selection five of garbage collectors [11, 12]:

1. Serial Garbage Collector (3 385 lines of code)
2. Parallel Garbage Collector (20 510 lines of code)
3. G1 – Generational Garbage Collector (50 274 lines of code)
4. The Z Garbage Collector (24 315 lines of code)
5. Shenandoah Garbage Collector (33 532 lines of code)

[3] git ls-files | xargs cat | wc -l on commit eccdd8e60399a4b0db77b94e77bb32381090a5c6

The serial, parallel and generational garbage collector are all three different implementations of generational collectors. The serial garbage collector will run serially along with the mutator and thus all mutator execution is paused and no advantage of multi-core processors are utilized. The parallel garbage collector still pauses the mutator during the entire garbage collection but uses several garbage collector threads to take advantage of multi-core processors to process garbage faster. G1, the default garbage collector since JDK 9, pauses the mutator in some phases, but others are done concurrently with the mutator, such as walking the object graph to find all living objects. Since these three, perform relocation when the mutator is paused, the most compact way of storing the forwarding information is using the old object's space.

The Z Garbage Collector (ZGC) and Shenandoah differ from the first three collectors, by being single generation and more importantly, by performing relocation concurrent with mutators. Both were designed with low pause times and in mind. ZGC and Shenandoah have significantly lower pause times than the other collectors [13, 14]. Since ZGC and Shenandoah do concurrent relocation, they need to store forwarding information in an apt way.

2.5 The Z Garbage Collector

The Z Garbage Collector (ZGC), is a concurrent and parallel garbage collector, designed around three goals [13]:

1. pause times should not exceed 10 ms
2. pause times should not increase with heap or live-set size
3. it should be able to handle heap sizes from hundreds of megabytes up to several terabytes

Memory allocation in ZGC is done using pages and bump pointer allocation [2, zPage.inline.hpp:228-243, zPage.hpp:42]. There exists three types of pages: small, medium and large [2, zGlobals.hpp:54-56], as depicted in Table 2.1. Since objects greater than 4MB get their own page, no bump pointer allocation is needed for large pages [15].

Page Type	Page Size	Object Size
Small	2MB	[0, 256]KB
Medium	32MB	(256KB, 4MB)
Large	≥4MB	≥4MB

Table 2.1: Page sizes in ZGC. (Figure taken from [15]). It should be noted that this table was correct in previous versions of ZGC (and for the paper that this table is reproduced from), but the commit [2] that this thesis builds upon on dynamically changes the size of medium pages depending on the machine. The table should, therefore, be viewed as an approximation of what range the pages will span and the size of the objects that they will contain.

An important design point in ZGC is its construction around load barriers rather than write barriers. In run-time systems, a barrier is some code triggered at some event [1]. As mentioned in Section 2.2, write barrier typically trigger on a write to a field, and a load barrier when a pointer is loaded to the stack. A barrier typically has a common fast path and a less common slow path [1]. As depicted in Figure 2.5, the fast path of the ZGC load barrier is just returning the read value whereas the slow path will do a bit more work and will, therefore, be slow in comparison.

One common optimization in modern CPUs is speculative execution and branch prediction [16]. Loading memory takes many CPU cycles and we want to spend as much as time as possible doing something (instead of waiting on memory loads). Speculative execution and branch prediction uses time spent on waiting on memory loads to execute a branch of code, which result may or may not be used, since the predicted branch to execute may be incorrect [16, 17]. Due to accuracy of modern branch predictors, the branch prediction will almost always correctly predict the fast path, reducing the effective cost of having a load barrier.

```
uintptr_t barrier(uintptr_t addr) {
  if (is_good_or_null(addr)) {
    // fast path
    return addr;
  } else {
    // slow path
    uintptr_t good_address = look_up_new_address(addr)
    return good_address;
  }
}
```

Figure 2.5: The ZGC load barrier executed when a pointer is loaded from the stack to the heap.

The load barriers in ZGC allow mutators to detect that a pointer is effectively dangling because an object has been moved as part of garbage collection. This causes the load barrier to take the slow path which looks up the new address of the object from the forwarding table. To avoid taking the slow path of subsequent accesses, the old dangling pointer is updated with the new location (not shown in the figure). In ZGC, this is referred to as “self-healing”. This means that mutators will never observe object movement due to GC, which means that this design effectively decouples garbage collection threads and mutator threads. To further lower the overhead cost of load barriers, ZGC only allows object relocation to happen in specific phases, which reduces the number of times a load barrier may trigger the slow path for a particular reference to once per garbage collector cycle.

Memory deallocation in ZGC happens as a side-effect of defragmentation. At the end of a ZGC cycle, all objects on sparsely populated memory pages will be moved over to another page and the original page will be free’d [2, zRelocate.cpp:182-205].

A pointer’s *color* [2, zBarrier.inline.hpp:139-156]. determines whether the fast path or the slow path in a load barrier are taken. ZGC has two color categories: good and bad [2, zGlobals.hpp:80–89] and a pointer’s color is stored in the higher-order

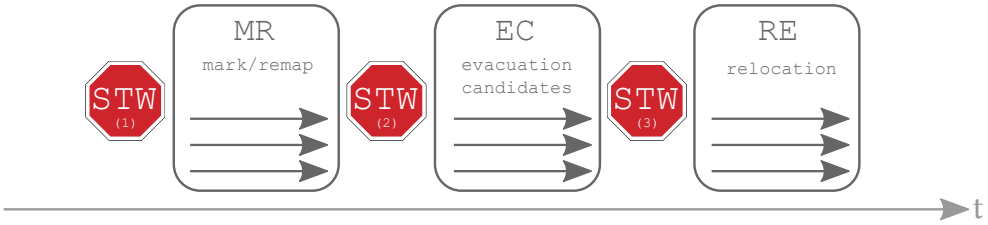


Figure 2.6: The ZGC cycle has three minimal stop-the-world pauses, aiming at most 10 ms in total per cycle, and three distinct concurrent phases. These are performed concurrently with the mutator and utilizes several garbage collector threads where possible (denoted by the three arrows). The arrows within the boxes represents that this phase is concurrent and parallel.

bits of the pointer addresses [2, zGlobals_x86.cpp:30–136].

A good colored pointer is guaranteed to point to the correct memory location but a bad colored pointer *may* point to an incorrect memory location. All threads are always in agreement on what color is considered good [2, zBarrierSet.cpp:91, zRelocate.cpp:50, zMark.cpp:135], which is handled through a brief stop-the-world pause. All pointers need to be verified at the beginning of each garbage collector cycle, therefore, at the beginning of each garbage collector cycle the color of all pointers is considered to be bad. This means that at the start of a garbage collector cycle, all references loaded to the stack will trigger the load barrier's slow path meaning all dangling pointers due to object relocation will be trapped.

2.5.1 The ZGC cycle

As depicted in Figure 2.6 one ZGC cycle has three pauses [2, zDriver.cpp:377–409] in which all mutator threads are stopped, denoted STW, and three concurrent phases: (1) mark and remap (MR), (2) select evacuation candidates (EC) and (3) relocation (RE).

The first STW, denoted as STW (1) decides which color is the good color by alternating two colours [15, 18]. This indirectly means that all pointers who were good in the the previous cycle are now bad, and will trigger the slow path on first load during the next cycle. In the first concurrent phase, the mark and remap phase (MR), object graph traversal is performed on all roots to identify all live objects. If the garbage collector finds any pointer with a bad color, the pointer will be updated to the current address and given the color. Pages are selected for relocation based on the density of their live population. To be able to decide if a page is sparsely populated, the size of all living objects on a page is recorded during the MR phase. STW (2) is needed as a synchronization point to ensure that the object graph traversal is completed. The EC phase may then begin, which will select all pages that have "too few" living objects. The selected pages are referred to as the *relocation set*. At the end of EC, a final STW (3) begins[4]. During the STW (3) pause, the good color will change which will make all pointers in the heap invalid. Mutators will, therefore, take the slow path, the first time any pointer is loaded. A relocation will be preformed if the object is living on a page that was selected as an eviction candidate otherwise a remap is preformed. After the final STW the RE phase takes place, where garbage collector threads will relocate all live objects in EC to new pages [2,

[4] During STW (2) and STW (3) reference processing is also performed. Reference processing is about handling non-strong reference types, which is an API through which a Java developer may interact with the GC. This is orthogonal to the work in this thesis, and we we refrain from discussing it further.

from	to
0x0	0xf0
0x12	0xf12
...	...

Figure 2.7: Example of a forwarding table for a specific memory page. Each row is an entry mapping a *from* address on the page to a *to* address on one x or more other pages.

zDriver.cpp:377–409].

Mutators and garbage collector threads may contend on relocating the same objects. This is rare for most applications and is handled by introducing a linearization point on the insertion in the forwarding table [2, zForwarding.inline.hpp:137-159]. When failure to insert a new object in a forwarding table, the failing thread undoes the allocation and simply reads the existing entry in the forwarding table. Because of this design, a relocated object's location will depend on what thread won the race to relocate it – a mutator or a GC thread.

2.5.2 Using a Forwarding Table to Keep Track of Relocated Objects

ZGC uses a forwarding table to store the new address of relocated objects. To this end, each page selected during EC will have an associated forwarding table, as depicted in Figure 2.7, which maps the old address to the new address [2, zForwarding.hpp, zForwardingEntry.hpp]. Any access to an address that has a bad color must check if there is an associated forwarding table to that page [2, zHeap.inline-.hpp:98–102, 118] and if so, then we must perform a relocation and/or remap the stale address to the new location.

As depicted in Figure 2.8, ZGC structures the forwarding information at three levels: a forwarding table (`ZForwardingTable`) for all relocated pages, forwarding information for a single page (`ZForwarding`) and forwarding information for an object in a page (an entry in `ZForwardingEntry`). The forwarding table is used to look-up if there are any forwarding information for a single page. The forwarding information about the single page is used to map the old address to the new address.

ZGC stores only the offset of the old address in the from page, since the full address can be calculated as the sum of the offset and the from page base address [2, zForwardingEntry.hpp:31–46]. A third column called *populated* tracks if the row is in use, and is thus able to differentiate between `NULL` and an entry that has not been updated [2, zForwardingEntry.hpp:31–46]. In the pathological case, an allocate attempt resulting in out-of-memory would result in a `NULL` address. Thus that bit is needed to be certain that the field has been updated.

2.5.3 Why Forwarding Table Has Large Off-Heap Memory Overhead

The `ZForwardingTable` is implemented using a map data structure, where the hash function provides constant time lookup. Performing modulo, a mod b, is usually an expensive operation but may be implemented efficiently with a bitwise AND operation if b. Therefore, to facilitate this, the amount of `ZForwardingEntry` is based on the number of live objects on the page times 2 and rounded up to the nearest power of two [5]. While it is computationally efficient, it has according from

[5] `nentries = round_up_power_of_2( page->live_objects() * 2)`

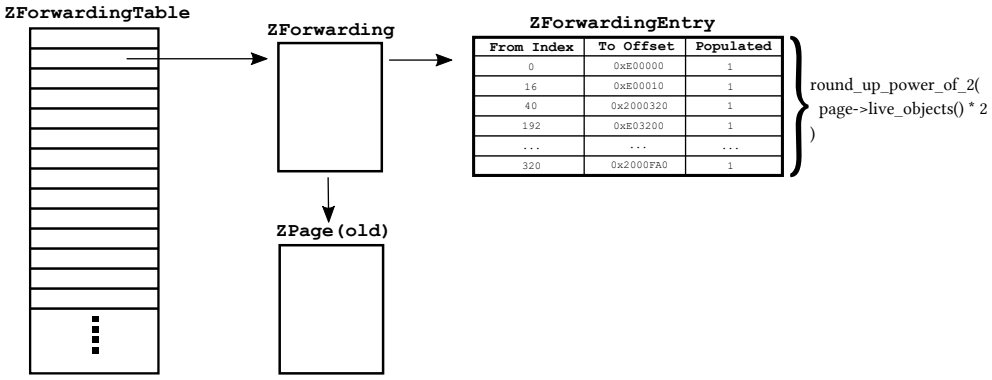


Figure 2.8: Overview of the forwarding information scheme in ZGC. The arrow represents a pointer, e.g. `ZForwardingTable` has a pointer to `ZForwarding` that has a pointer to the old page (`ZPage (old)`) and to the table containing all entries (`ZForwardingEntry`).

a code comment in the source code of ZGC on average only a load factor of 50% [2, zForwarding.cpp:32–35]. This and the inherent fact that it needs to store an entry for each object is the cause as to why the off-heap memory overhead can be large.

Assume that a page is selected for relocation and that it would fit n amount of the smallest allocatable object, which in ZGC is of size 16 bytes. A theoretical worst-case occurs when this page contains $n-1$ objects, since the number of entries is determined by first doubling the number of live objects and then round up to the nearest power of two. This results in needed as much space for the entires as for the entire page, *i.e.,* a 100% memory overhead.

2.5.4 Live Map

During object graph traversal in the marking phase, MR in Figure 2.6, the address of each object that is live, is stored in a live map. The live map is a compact data structure implemented using a bit map to represent a live object starting at a particular address (or not). Each bit in the live map represents a word, 8 bytes in ZGC. To cover the entire addressable space on a small page (2 MB), we thus need 242 144 bits. Each page has its dedicated live map and this can be traversed to get the address of all live objects residing on that page, which is a more efficient way of finding all live objects as opposed to doing a full object graph traversal of all roots.

Let us consider Figure 2.9. For simplicity, assume that the page's start address is `0x0`. To calculate the address of the first living object we should take the start address `0x0` and multiply with the size of a word times the index we encountered: $\texttt{0x0} + \texttt{0x8} \cdot 1 = \texttt{0x8}$.

Figure 2.9: Example of entries in the live map.
Assuming that the first entry is recorded at the most left hand side.

We have now introduced garbage collection in general and ZGC in particular. In the next chapter, we propose a new design for a compressed forwarding table that

trades memory for additional computation on forwarding table lookup. Some similarities with the new design is found in the garbage collector called Compressor [19]. It seems that the high level design is similar to ours, but we differ on optimization and contextual decisions.

Chapter 3

Design of a Compressed Forwarding Table

The ZGC way of storing forwarding information $A \rightarrow B$ for each relocated object A to its new location B as a separate entry in a hash table is computationally efficient. Each entry needs approximately 128 bytes (64 bytes for A respectively B), but to improve lookup performance, hash tables grow in size in such a way that memory is allocated for unused entries, making the effective cost of each entry considerably higher.

This chapter proposes a new design for storing forwarding information on a per-page level in a way that allows B to be calculated on-the-fly. The design results in a compressed forwarding table with a worst-case memory overhead of $< 3.2\%$. This is achieved since we are targeting to use only 64 bits per entry in the table, additionally this allows for good cache performance. As such, this design immediately solves the problem of different programs, or different runs of the same program, or even different phases of the same program, giving rise to considerably different memory overhead. Chapter 5 investigates the performance impact of this design, *e.g.,* as each lookup requires additional computation. These changes will become clearer further on.

In this chapter it is important to keep in mind that when referring to *old pages* and objects living on old pages, this also implies that they are in the relocation set. As mentioned in Section 2.5.1, pages/objects in the relocation set are subject to relocation. All living objects on old pages will be moved to a *new page* allowing all old pages to be free'd. In ZGC, there is no way to a priori calculate the new location of an object on an old page as relocations performed by mutators move objects into their own allocation buffers. A key change in the new design is "deterministic relocation" which means that an object's forwarding address is constant, regardless of who performs the relocation. Storing forwarding information about each object explains why the high off-heap memory overhead can become large. Shenandoah, ART and ZGC allows objects to be relocated non-deterministic addresses[1]. This means that they require to store forwarding information on the per-object level. The new design of "compact forwarding information" is not specific to ZGC and should apply to other garbage collectors such as Shenandoah or ART.

[1)] While the address range that objects will be relocated to might be known, their design does not support knowing the exact address that the object will get.

For the sake of simplicity, this chapter is written as if there were only one old page and one new page.

3.1 Deterministic Concurrent Relocation

A key component of the new design is making relocation deterministic in the sense of an relocating an object on an old page to a predetermined address regardless of who is performing the relocation: a GC thread, or a mutator. This voids the need to store each object's forwarding address.

The deterministic relocation scheme moves all old objects in the address order on the old page to the new page, as depicted in Figure 3.1. To calculate the forwarding address B of an object A on a page P, we must calculate the sum of the sizes of all live objects on P upto A. This gives us the object's offset on the page to which it will be relocated. Figure 3.2 shows the code for calculating the new forwarding address for a given old address in pseudo C++.

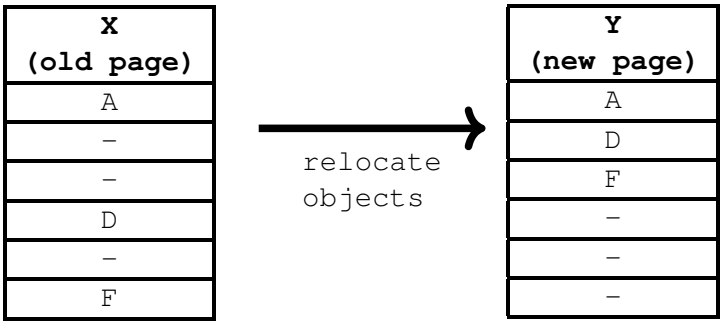


Figure 3.1: Example of keeping the order of the objects after relocation.

```
uintptr_t new_address(uintptr_t old_address) {
  size_t live_bytes_preceding_address =
    count_live_bytes_preceding(old_address); // O(n)

  return
    new_page()->get_start_address() +
    live_bytes_preceding_from_address;
}
```

Figure 3.2: Pseudo code of simple deterministic address calculation.

Naturally, the design above is not computationally efficient as it needs to repeatedly calculate the size of the same object to perform multiple relocations. We avoid this by dividing each page into smaller fixed-size chunks and for each chunk keeping track of the sum of the sizes of all living objects in the preceeding chunks. Logically, this has the same structure as a forwarding table, but the number of entries is much smaller as each chunk ranges over multiple objects.

With this in place, to calculate $A \rightarrow B$, we first calculate the chunk in which A resides ($\mathcal{O}(1)$), to lookup the preceeding live bytes *PLB* in constant time. Then, we only need to count the live bytes preceeding A inside the chunk itself, *CLB*, ($\mathcal{O}(n)$). Then, B can be calculated as *PLB* + *CLB*. Figure 3.3 shows this in pseudo C++. The new algorithm is linear in the size of a chunk, as opposed in the size of a page.

By performing the precalculation of preceeding live bytes for each chunk in the garbage collector threads, this cost is hidden from the mutator threads, provided that there are available cores in the machine that are free to do garbage collector

```
uintptr_t new_address(uintptr_t old_address) {
  size_t live_bytes_preceding_entry_address =
    accumulated_live_bytes(old_address); // O(1)

  size_t live_bytes_within_entry_address =
    count_live_bytes_within_entry(old_address); // O(n)

  return
    new_page()->get_start_address() +
    live_bytes_preceding_entry_address +
    count_live_bytes_within_entry;
}
```

Figure 3.3: Pseudo code of optimized deterministic address calculation.

work. To find a good balance between computational efficiency and memory usage it is imperative to find a good chunk size.

Finally, when relocation is performed, we relocate an entire chunk, not single objects, since we need to deal with contention between garbage collection threads and mutator threads, described in more details below.

3.2 Dealing with Contention on Relocation

As mutators are running concurrent with GC threads during relocation, a mutator might access objects on a page that is slated for relocation. If the GC threads have already relocated this page, remapping occurs in the slow path of the load barrier. However, if GC thread have not yet relocated the object, the mutator will perform the relocation before going forward. This might cause GC threads and mutators to occasionally compete on relocating the same object.

In ZGC, such competition might lead to two copies of the same object ending up at different locations, but only one of the locations will be added to the forwarding table, and become the new address of the object. In the new design, even though both GC threads and mutators would relocate the object to the same place, we could end up in an incorrect state in the program: Consider the object A and suppose that A is relocated by the mutator thread, which subsequently writes to a field in A after the relocation. If a garbage collector thread is performing the relocation concurrently with the mutator, this could accidentally overwrite the updated field in A, causing the update to be lost.

The problem is solved by preventing an object from being relocated more than once, which can be achieved by introducing an additional synchronization step in the relocation algorithm. Pseudo code for the synchronziation is found in Figure 3.4. As can be seen in that figure, before a relocation of an object can begin, we need to find its compact entry and then try to take a lock for the object's address. Since threads are competing to relocate objects someone else might have already performed relocation, so when we acquired the lock we must begin with checking that this object still needs relocation. If the object still needs relocation, then relocation of the entire entry is performed and the pseudo code for relocating all objects on the compact entry is found in Figure 3.5. We do not want to copy an object more than once, so all objects within a compact entry are moved at a time, as this allows us to store whether the objects have been moved or not using only one bit in the

compact entry. It would have been possible to store whether an object has been moved or not in a more fine-grained fashion, but that would have increased the memory usage and probably incurs poor cache performance, because it would be hard to fit into our target of only using 64 bits per entry.

The overhead and performance impact of the additional synchronization step is inherently tied to its granularity. In one end of the spectrum, we could have one global lock, forcing all relocations to synchronize. This would be highly inefficient, since most relocations are unrelated and thus have no contention problem. In the other end of the spectrum we could synchronize each individual object relocation. This would create a large memory overhead and while it does happen that the mutator and the garbage collector thread compete to relocate the same object, it is not the common case. Thus, a middle-grained synchronization scheme is likely the better choice.

We suggest that the address space is divided into k subspaces and that we have a lock for each subspace. The number of subspaces will be decided by multiplying the number of cores in the machine by two and rounded up to the nearest power of two. Modulo, a mod b, is usually an expensive operation but may be implemented efficiently with a bitwise AND operation if b is a power of two. We are assuming that the number of subspaces will be large enough as to not cause too much contention for the locks. If the design is successful the number of subspaces is something that could be further examined and evaluated, since this assumption for approximating the number of subspaces may be incorrect.

3.3 Calculating Accumulated Live Bytes for Chunks

When calculating the new address for object F in Figure 3.1, we need to know the size of all preceding live object on the old page and the starting address of the new page. A naive approach to get the size of all preceding living objects would be to traverse the live map described in Section 2.5.4 and get the size of the object from its metadata recording.

The two main issues with this naive approach is first, the obvious problem of needing to scan the large sections of the page several times, for each object to be moved. The second, more subtle problem, is that to be able to ask the object about its size we can't free the old page until every pointer has updated its pointer to the new address. The current behavior in ZGC is to free the old page as soon as all objects have been relocated, so in comparison to that, it would increase the floating garbage.

A better design is to divide the page into Q amount of compact entries, divisible by 2^n, $n \in \mathbb{N}$. Let each compact entry represent the amount of previously living objects *before* that compact entry. To get the size of previously living objects you would then use the associated compact entry and scan the live map for the addresses who is not covered by the compact entry. This significantly reduces the search time and allows for freeing the old page as soon as all objects have been relocated, at the cost of some space. An example of dividing a page into compact entries is depicted in Figure 3.6.

The more fine-grained the compact entries are, the less you would have to search in the live map. The most extreme case is to have as many compact entries as there are addresses, which then would essentially be the same as the forwarding table from the original ZGC, described in Section 2.5.2.

```
uintptr_t relocate_object(ZCompact* compact,
  uintptr_t from_addr) {
  ZCompactEntry *c = compact->find(from_addr);

  lock_map.lock(from_addr);
  if (c->copied()) {
    uintptr_t to_addr = compact->to_offset(from_addr, e);
    lock_map.unlock(from_addr);
    return to_addr;
  }

  const uintptr_t to_addr =
    relocate_object_inner(compact, from_addr);
  lock_map.unlock(from_addr);

  return to_addr;
}
```

Figure 3.4: Pseudo code for initiating relocation of an object.

```
uintptr_t relocate_object_inner(ZCompact* compact,
  uintptr_t from_addr) {
  ZCompactEntry* c = compact->find(from_addr);
  const uintptr_t to_addr = compact->to_addr(from_addr, c);

  if (c->copied()) {
    // Already relocated
    return to_addr;
  }

  // Reallocate all live objects within the compact entry,
  // from_addr is one of these objects
  for (from_addr_entry in c) {
    uintptr_t to_addr_entry =
      compact->to_addr(from_addr_entry, c);

    object_copy(from_addr_entry,
      to_addr_entry,
      object_size(from_addr_entry));
  }
  c->set_copied();

  return to_addr;
}
```

Figure 3.5: Pseudo code for relocation of an object that exists on a compact entry.

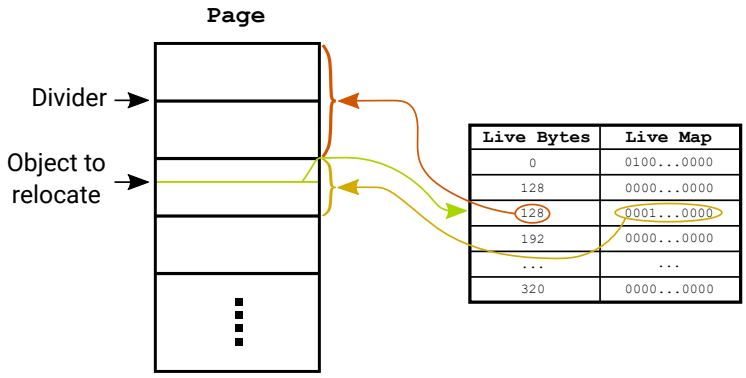


Figure 3.6: Using pre-calculated accumulated live bytes speeds up forward address calculation. Page dividers divide pages into chunks and each chunk has a corresponding row in the compact forwarding table. This row holds the amount of live bytes in the preceding chunks, and a live map for all the objects on the chunk. The green line and arrow shows an object to be relocated and its corresponding row in the forwarding table. The red arrow shows what chunks the accumulated live bytes summarize. To calculate the forwarding address, we only need to look at the row pointed to by the green allow as all preceding rows are summarizes by the live bytes. As depicted by the yellow arrow, the live map covers the chunk where the object we are relocating resides. Thus, in addition to the live bytes, we must search this map to calculate the object's offset into the chunk.

3.4 Putting It All Together

Let us construct a compact table where the first column is accumulated live bytes and the second column is a live map, where each bit represents a word. For memory efficiency and with regards to memory alignment, one entry fits in 8 bytes (64 bits) Using 32 bits, each live map can thus describe 256 bytes in a page. The bit layout for each entry is found in Figure 3.7. Each row is said to span a *fragment* of the page and to be a *compact entry*. Then let the n:th row's accumulated live bytes be the amount of all live bytes of up to $n-1$ rows.

To find the amount of live bytes for preceding objects you would then find the corresponding compact entry and then sum the accumulated live bytes and scan the live map to find all living objects residing on the compact entry.

To be able to free the old page as soon as we have relocated all objects, the live map is extended to not only store the start of the object, but also the end. Since we only need to know the size of all *preceding* objects, storing the end bit of objects spanning over the compact entry limit is not needed, because that implies that it is the last object on the compact entry.

An example of how the compact table is populated with liveness information and accumulated live bytes is given in Figures 3.8–3.10. Figure 3.8 depicts the entries from an initialized state and then populating them with liveness information such that accumulated live bytes calculation can begin. Additionally, this figure also shows that the first entry for the accumulated live bytes is always zero since the first entry has no previous entries to refer to. Figures 3.9–3.10 steps through each

entry and exemplifies how accumulated live bytes are calculated. A third column, address range, is added in the figures to help a human to read it, but it is not needed for the implementation. This column represents what addresses the live map for that particular row is spanning.

Let us show the use of these tables through and example. Let's say we want to calculate the forwarding address of an object living on address `0x188`, in Figure 3.11a and that the start address of the new page is `0x2000000`. This object would then correspond to the blue bit in the figure. To get the new address for the object at `0x188` you would therefore sum the following in Figure 3.11b.

3.5 Structural Code Comparison With the Original Solution

ZGC stores forwarding information in a global lookup table, `ZForwardingTable`. Each page selected for relocation would have a corresponding `ZForwarding` entry in the global `ZForwardingTable`. Furhter, each `ZForwarding` has as many entries as needed for each relocated object, stored using `ZForwarding-Entry` in a consecutive array. This is described in more detail in Section 2.5.2.

To depart as little as possible from the existing structure of the code, forwarding information in our implementation of the design exists at three different levels. There is a global lookup table, `ZCompactForwardingTable` with a `ZCompact-Forwarding` for each page that has been selected for relocation. `ZCompact-Forwarding` stores all per-page information needed for relocation and a pointer to the array containing `ZCompactForwardingEntry`. Fig. 3.12 shows these designs back to back.

3.6 Pre-Committing Memory During Initialization

While it is obvious that `ZCompactForwarding` should store the page address of the selected page for relocation as the old address, destination addresses for new page(s) is more tricky. To minimize fragmentation, objects on old pages should be relocated to new pages without creating new fragmentation. Recall that the original ZGC implementation reduces fragmentation after relocation, so this behavior should be kept. Increasing fragmentation increases the risk of out-of-memory errors despite available memory. This is due to the bump pointer allocation scheme used in ZGC. During initialization, the live map is traversed when it is transcribed into each `ZCompactForwardingEntry`, as a result this makes it an easy point to calculate if objects would fit into the new page. To simplify calculation and assignment of new page(s), these are allocated and pre-comitted during initialization. All `ZCompactForwarding` have their page addresses and accompanying offset installed succeeding the initialization.

3.7 The Impact of the New Design

3.7.1 Upper Bound of Memory Overhead

As we have already stated, the memory overhead of the new design will never exceed 3.2%. Here is how we have arrived as this number[2] as the maximal overhead.

[2] The large memory cost comes from all entries. It should be noted that the size of the additional meta data that is needed once per page is omitted from this calculation. It is assumed to not be more than 1KB which would increase the overhead cost by 0.0005% and is thus negligible.

Let S be the size of a chunk in bytes, and B be the number of bytes whose forwarding information is covered by one chunk. Note that both S and B are constant and do not change *e.g.*, with the page size, or number of relocated objects. Thus, the memory overhead needed to store forwarding information is S/B. In our implementation, each chunk covers 256 bytes of the heap and the size of a chunk is 8 bytes (64 bits). This means that for the selected design parameters the following memory overhead is:

$$\frac{8}{256} = 3.125\%$$

Naturally, if only 10% of all pages of the heap are relocated, we only need forwarding information for one tenth of the heap, meaning that the effective overhead is 0.3125%. § 3.7.2 describes how we used simulation to explore the effective overhead before we had a proper implementation in place.

3.7.2 Approximating Effective Overhead through Simulation

We approximated the effective overhead or our design by inserting telemetry into the source of an otherwise unmodified OpenJDK, adding only 20 lines of code. All (small) objects in the relocation set times the size of compact entries and the size of the forwarding entries is logged as can be seen in Figure 3.15. Medium sized objects were omitted since they would be very few (if any), in the selected benchmarks. We run out instrumented OpenJDK on two of the DaCapo benchmarks (h2 and tradebeans) and the synthetic worst case scenario called BigRamTester[3]. Fig. 3.13 and Fig. 3.14 summary the simulation and show that the new memory overhead varies less, with regards to maximum memory usage and the memory overhead is significantly less in applications that do a lot of relocation (BigRamTester). Furthermore, while the upper limit for the memory usage of the new design is found to be 3.2%, the simulations shows that for BigRamTester and h2 the memory usage is significantly lower than that, 2.6% and 1.2% respectively.

[3] Source code available as an attachment from `https://bugs.openjdk.java.net/browse/JDK-8152438`.

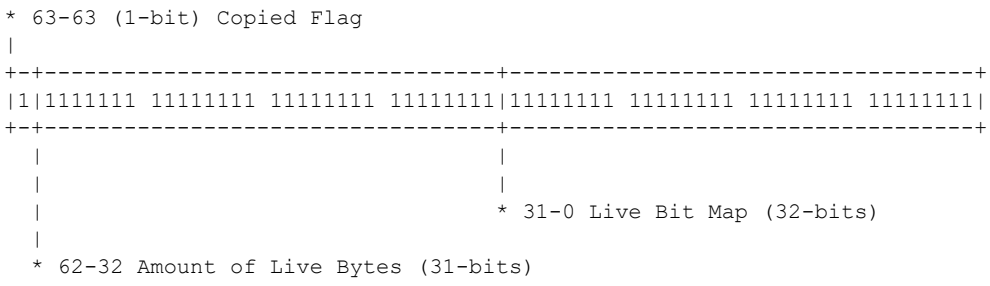


Figure 3.7: Bit layout for a compact entry.

Accumulated live bytes	Live map	(Address range)
0	0000 0000 0000 0000 0000 0000 0000 0000	0x0 – 0xFF
0	0000 0000 0000 0000 0000 0000 0000 0000	0x100 – 0x1FF
0	0000 0000 0000 0000 0000 0000 0000 0000	0x200 – 0x2FF
0	0000 0000 0000 0000 0000 0000 0000 0000	0x300 – 0x3FF
0	0000 0000 0000 0000 0000 0000 0000 0000	0x400 – 0x4FF
0	0000 0000 0000 0000 0000 0000 0000 0000	0x500 – 0x5FF
...	...	...
0	0000 0000 0000 0000 0000 0000 0000 0000	0x1FFEFF – 0x1FFFFF

(a) Starts with uininitalized compact entries

Accumulated live bytes	Live map	(Address range)
0	0000 0000 0000 0000 0000 0000 0000 0000	0x0 - 0xFF
0	0001 1000 0000 0000 1000 0000 0000 0100	0x100 - 0x1FF
0	0000 0000 0000 0000 0000 0000 0000 0000	0x200 - 0x2FF
0	0000 0000 0000 0000 0000 0000 0000 0000	0x300 - 0x3FF
0	0011 0010 0000 0000 0001 0000 0000 0000	0x400 - 0x4FF
0	0000 0000 0000 0000 0000 0000 0000 0000	0x500 - 0x5FF
...	...	...
0	0000 0000 0000 0000 0000 0000 0000 0000	0x1FFEFF - 0x1FFFFF

(b) Copying liveness and size information into the live map. Green bit means that an object that is live starts here and orange means that the object ends on this address.

Accumulated live bytes	Live map	(Address range)
0	0000 0000 0000 0000 0000 0000 0000 0000	0x0 - 0xFF
0	0001 1000 0000 0000 1000 0000 0000 0100	0x100 - 0x1FF
0	0000 0000 0000 0000 0000 0000 0000 0000	0x200 - 0x2FF
0	0000 0000 0000 0000 0000 0000 0000 0000	0x300 - 0x3FF
0	0011 0010 0000 0000 0001 0000 0000 0000	0x400 - 0x4FF
0	0000 0000 0000 0000 0000 0000 0000 0000	0x500 - 0x5FF
...	...	...
0	0000 0000 0000 0000 0000 0000 0000 0000	0x1FFEFF - 0x1FFFFF

(c) The accumulated live bytes is always 0 for the first row, since there are no previous rows.

Figure 3.8: How the compact entries are filled step-by-step. (a) We go from an uninitialized state, to copying liveness information and begins calculating the accumulated live bytes.

Accumulated live bytes	Live map	(Address range)
0	0000 0000 0000 0000 0000 0000 0000 0000	0x0 - 0xFF
0	0001 1000 0000 0000 1000 0000 0000 0100	0x100 - 0x1FF
0	0000 0000 0000 0000 0000 0000 0000 0000	0x200 - 0x2FF
0	0000 0000 0000 0000 0000 0000 0000 0000	0x300 - 0x3FF
0	0011 0010 0000 0000 0001 0000 0000 0000	0x400 - 0x4FF
0	0000 0000 0000 0000 0000 0000 0000 0000	0x500 - 0x5FF
...	...	...
0	0000 0000 0000 0000 0000 0000 0000 0000	0x1FFEFF - 0x1FFFFF

(a) Accumulated live bytes is updated to 0 since previous accumulated live bytes is 0 and the size of all living objects on the previous compact entry was 0.

Accumulated live bytes	Live map	(Address range)
0	0000 0000 0000 0000 0000 0000 0000 0000	0x0 - 0xFF
0	0001 1000 0000 0000 1000 0000 0000 0100	0x100 - 0x1FF
128	0000 0000 0000 0000 0000 0000 0000 0000	0x200 - 0x2FF
0	0000 0000 0000 0000 0000 0000 0000 0000	0x300 - 0x3FF
0	0011 0010 0000 0000 0001 0000 0000 0000	0x400 - 0x4FF
0	0000 0000 0000 0000 0000 0000 0000 0000	0x500 - 0x5FF
...	...	...
0	0000 0000 0000 0000 0000 0000 0000 0000	0x1FFEFF - 0x1FFFFF

(b) Accumulated live bytes is updated to 128 since previous accumulated live bytes is 0 and the size of all living objects on the previous compact entry was 128.

Accumulated live bytes	Live map	(Address range)
0	0000 0000 0000 0000 0000 0000 0000 0000	0x0 - 0xFF
0	0001 1000 0000 0000 1000 0000 0000 0100	0x100 - 0x1FF
128	0000 0000 0000 0000 0000 0000 0000 0000	0x200 - 0x2FF
128	0000 0000 0000 0000 0000 0000 0000 0000	0x300 - 0x3FF
0	0011 0010 0000 0000 0001 0000 0000 0000	0x400 - 0x4FF
0	0000 0000 0000 0000 0000 0000 0000 0000	0x500 - 0x5FF
...	...	...
0	0000 0000 0000 0000 0000 0000 0000 0000	0x1FFEFF - 0x1FFFFF

(c) Accumulated live bytes is updated to 128 since previous accumulated live bytes is 128 and the size of all living objects on the previous compact entry was 0.

Figure 3.9: How the compact entries are filled step-by-step. (b) We continue to calculate accumulated live bytes.

Accumulated live bytes	Live map	(Address range)
0	0000 0000 0000 0000 0000 0000 0000 0000	0x0 - 0xFF
0	0001 1000 0000 0000 1000 0000 0000 0100	0x100 - 0x1FF
128	0000 0000 0000 0000 0000 0000 0000 0000	0x200 - 0x2FF
128	0000 0000 0000 0000 0000 0000 0000 0000	0x300 - 0x3FF
128	0011 0010 0000 0000 0001 0000 0000 0000	0x400 - 0x4FF
0	0000 0000 0000 0000 0000 0000 0000 0000	0x500 - 0x5FF
...	...	...
0	0000 0000 0000 0000 0000 0000 0000 0000	0x1FFEFF - 0x1FFFFF

(a) Accumulated live bytes is updated to 128 since previous accumulated live bytes is 128 and the size of all living objects on the previous compact entry was 0.

Accumulated live bytes	Live map	(Address range)
0	0000 0000 0000 0000 0000 0000 0000 0000	0x0 - 0xFF
0	0001 1000 0000 0000 1000 0000 0000 0100	0x100 - 0x1FF
128	0000 0000 0000 0000 0000 0000 0000 0000	0x200 - 0x2FF
128	0000 0000 0000 0000 0000 0000 0000 0000	0x300 - 0x3FF
128	0011 0010 0000 0000 0001 0000 0000 0000	0x400 - 0x4FF
256	0000 0000 0000 0000 0000 0000 0000 0000	0x500 - 0x5FF
...	...	...
0	0000 0000 0000 0000 0000 0000 0000 0000	0x1FFEFF - 0x1FFFFF

(b) Accumulated live bytes is updated to 256 since previous accumulated live bytes is 128 and the size of all living objects on the previous compact entry was 128.

Accumulated live bytes	Live map	(Address range)
0	0000 0000 0000 0000 0000 0000 0000 0000	0x0 - 0xFF
0	0001 1000 0000 0000 1000 0000 0000 0100	0x100 - 0x1FF
128	0000 0000 0000 0000 0000 0000 0000 0000	0x200 - 0x2FF
128	0000 0000 0000 0000 0000 0000 0000 0000	0x300 - 0x3FF
128	0011 0010 0000 0000 0001 0000 0000 0000	0x400 - 0x4FF
256	0000 0000 0000 0000 0000 0000 0000 0000	0x500 - 0x5FF
...	...	...
256	0000 0000 0000 0000 0000 0000 0000 0000	0x1FFEFF - 0x1FFFFF

(c) Final result. Last entry of accumualted live bytes is still 256 bytes since there were no more living objects on this page.

Figure 3.10: How the compact entries are filled step-by-step. (c) The accumulated live bytes is calculated for all entries.

Accumulated live bytes	**Live map**	(Address range)
0	0000 0000 0000 0000 0000 0000 0000 0000	0x0 - 0xFF
0	0001 1000 0000 0000 1000 0000 0000 0100	0x100 - 0x1FF
128	0000 0000 0000 0000 0000 0000 0000 0000	0x200 - 0x2FF
128	0000 0000 0000 0000 0000 0000 0000 0000	0x300 - 0x3FF
128	0011 0010 0000 0000 0001 0000 0000 0000	0x400 - 0x4FF
256	0000 0000 0000 0000 0000 0000 0000 0000	0x500 - 0x5FF
...	...	...
256	0000 0000 0000 0000 0000 0000 0000 0000	0x1FFEFF - 0x1FFFFF

(a) Example of compact forwarding entries. The live map should be read from left to right.

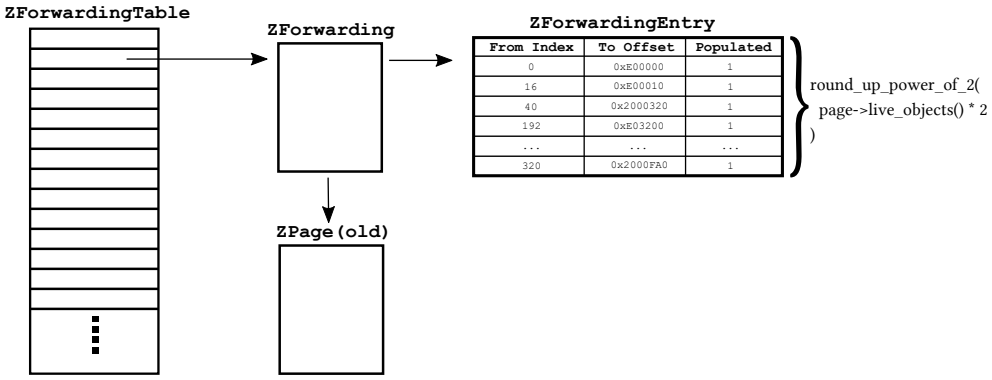
Address of the object starting at the blue bit: *start address of the new page + accumulated live bytes at row 2 + the size of all preceding objects on that compact entry*

=

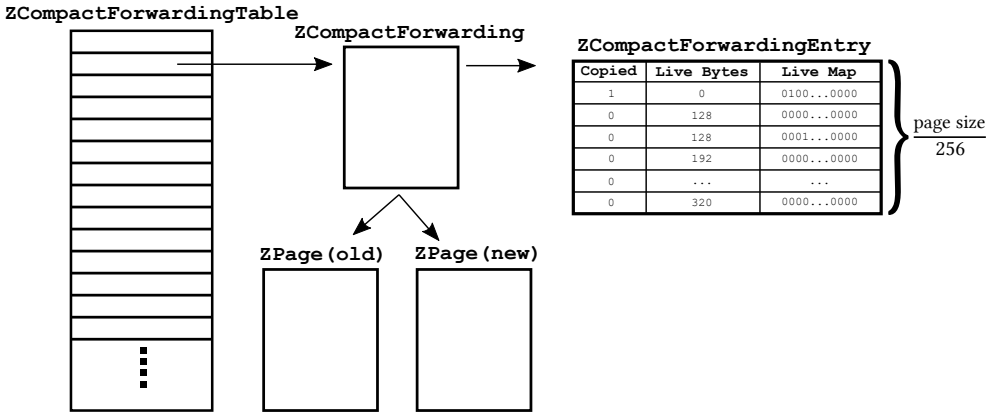
$$0x2000000 + 0x0 + 0x10 = 0x2000010$$

(b) Example of how an address can be calculated from the compact forwarding table

Figure 3.11: Example of how an address can be calculated from the compact forwarding table



(a) Forwarding table. Forwaring information is stored per object level. The amount of entires can become large since it is multiplied by two with the number of live objects and rounded up to the nearest power of two.



(b) Dividing page into compact entries. Forwarding information is stored per page level. The amount of entries is constant with regards to the page size.

Figure 3.12: Overview of the original and new forwarding information scheme. The arrow indicates a pointer.

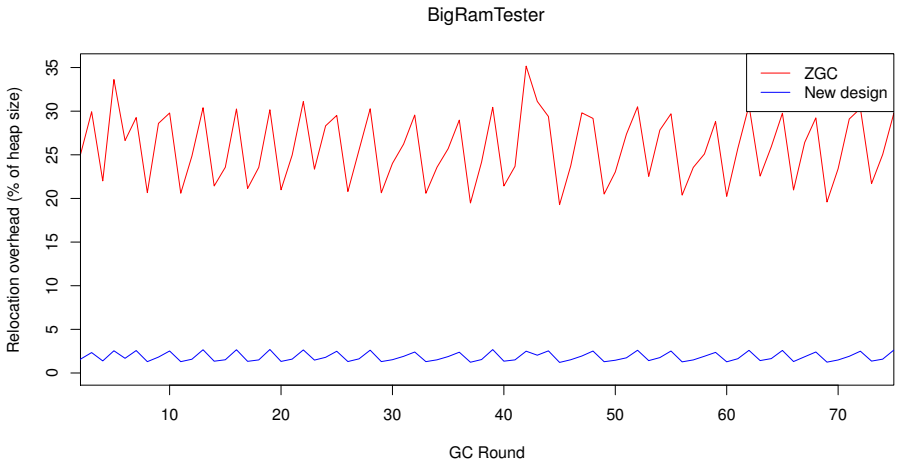


Figure 3.13: Memory overhead in ZGC vs the new design in BigRamTester. A peak value for the memory overhead for ZGC is obsereved to be 35% vs. 2.6% for the new design.

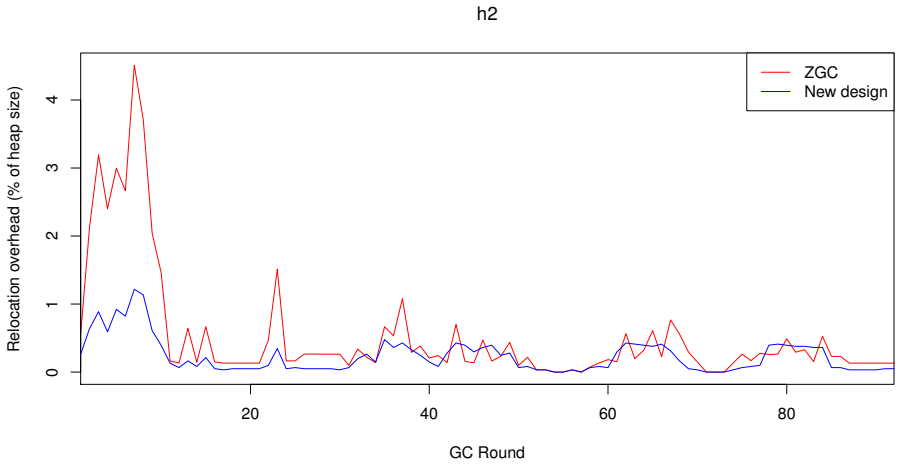


Figure 3.14: Memory overhead in ZGC vs the new design in h2. A peak value for the memory overhead for ZGC is obsereved to be 4.5% vs. 1.2% for the new design.

```
void ZRelocationSet::setup_relocation_set(ZPage* const* small,
  size_t nsmall) {
  size_t count_zforwardingentry = 0;

  // Setup small pages
  for (size_t i = 0; i < nsmall; i++) {
    _forwardings[j++] = ZForwarding::create(group1[i],
      &count_zforwardingentry);
  }

  size_t zforwarding =
    count_zforwardingentry * sizeof(ZForwardingEntry);
  size_t zcompactforwarding =
    nsmall * 8192 * sizeof(ZCompactForwardingEntry);
  // A small page has 8192 entries in the new design
  log_info(gc)("Total allocated ZForwardingEntry size: "
    SIZE_FORMAT, zforwarding);
  log_info(gc)("Total allocated ZCompactForwardingEntry size: "
    SIZE_FORMAT, zcompactforwarding);
}


ZForwarding* ZForwarding::create(ZPage* page,
  size_t* count_zforwardingentry) {
  const size_t nentries =
   round_up_power_of_2(page->live_objects() * 2);
   // ZGC rounds to a power of two for
   // computational efficency
  *count_zforwardingentry = nentries + *count_zforwardingentry;
  return new Array<ZForwarding(page), 1, ZForwardingEntry, nentries>;
}
```

Figure 3.15: Pseudo code of added telemetry for simulating the new design.

Chapter 4

Evaluation Methodology

Optimizations typically involves a memory–calculation trade-off. A classic example of this is memoization — keeping track of results of applications to allow recurring applications to be resolved in constant-time, at the additional cost of storing a map from arguments to results in memory [20], for each function. ZGC, Shenandoah and ART rely on storing forwarding information in a map-like fashion for fast lookup. In ZGC, forwarding information is stored in a map data structure in a contiguous address space, whereas in Shenandoah and ART, the map is distributed over pages selected for evacuation, with each entry inscribed in the space occupied by relocated objects.

By trading memory for some additional calculation-time, our approach is to implement a "compact forwarding table" which achieves low overhead by keeping the minimal information around that allows each lookup to calculate the unique forwarding address for each live object. By placing this information in an auxiliary data structure, we avoid the floating garbage problem of Shenandoah and ART due to delaying recycling of entire memory pages. Since we keep information to calculate forwarding addresses, instead of simply recording them, the memory costs does not increase with the number of relocated objects (although with the number of pages), avoiding the pathological case of ZGC.

When comparing garbage collectors, there are, on a high level, three important aspects to consider: pause time, space and throughput [1]. Pause times in ZGC are only effected by the root set and the new design does not introduce any new roots. If we generalize from our simulations (see Figures 3.13–3.14), we know that the memory overhead is stable and low, since it is independent from the amount of living objects. Usually when you give up memory, the computational cost increase. This means that it is important to understand what impact the new design has on the throughput.

We expect compressed forwarding information to lead to some performance degradation, since we are now calculating addresses on-the-fly. However, systems such as OpenJDK are complex, and our expectation may be incorrect. Furthermore, the amount of performance regression is hard to gague. To understand the performance implications, we can measure applications' execution times/benchmark scores. A goal guided by the author's intuition is to not have a more than 5% regression with respect to execution time or throughput related benchmark scores. Moreover, any change related to timing could change how the garbage collector behaves. This means that the size of the relocation set could change. To get a better understanding how the changes to the forwarding information effects OpenJDK

with ZGC, it is of interest to also measure the duration of the relocation phase and the size of the relocation set.

4.1 Measuring Throughput

To measure the application throughput, SPECjbb2015 and the DaCapo benchmark suite are used. SPECjbb2015 measures this using a internally defined benchmark score jOPS and the DaCapo suite measures application throughput in execution time. Both are commonly used in the research community when benchmarking JVM performance. For short-lived benchmarks in the DaCapo benchmark suite, the method to retrieve statistically significant results is described in Section 4.1.5.

The synthetic worst-case scenario, BigRamTester[1], presented first in Chapter 1 and later in Chapter 3 is important to guage the differences in the memory usage of the different solutions. BigRamTester however, does not perform any "meaningful" work. It essentially creates a big linked list and subsequently removes random elements from it (using several threads) and keeps doing this forever. So any results from these measurements would have minimal general applicability. For two reasons, we do not measure execution time for BigRamTester since it would provide little value into answering the question on what the performance impact would be of the new design.

[1] Source code available as an attachment from `https://bugs.openjdk.java.net/browse/JDK-8152438`.

4.1.1 DaCapo Suite

The DaCapo benchmark suite is a collection of carefully selected benchmarks to be used by the community that develops Java to benchmark the language implementation [21]. A benchmark is added to the suite if it is considered to offer a unique aspect with regards to measuring performance aspects of the JVM. Table 4.1 contains a list from the documentation from DaCapo, briefly describing each benchmark that were selected for benchmarking. We use DaCapo-9.12-bach-MR1, which is the latest stable release at the time of writing this. Since most of the benchmarks are rather short-lived, the method described in Section 4.1.5 needs to be used to retrieve statistically significant results. The benchmarks tradebeans and tradesoap could not run on the author's system due to application errors which we believe are caused by a concurrency problem that is reported in the issue tracking system at Github[2]. The benchmarks eclipse and batik could not run on newer JDKs due to dependencies issues, which is also reported in the issue tracking system at Github[3]. Finally, while tomcat did run, we were never able to get a correct validation of the benchmark, neither with original OpenJDK nor using our new implementation, so it also had to be excluded.

[2] `https://github.com/dacapobench/dacapobench/issues/99`

[3] `https://github.com/dacapobench/dacapobench/issues/175`

The latest stable release of DaCapo is 11 years old (2009), but is still used in the community and is, therefore, included as to provide credibility and ability to compare results against other garbage collectors. There exists a release candidate [4] that adds new benchmarks, offering additional unique aspects for benchmarking. The benchmark BioJava, a library for processing biological data [5], were selected from these new additions. GraphChi is excluded due to time constraints and the other ones are excluded since they were not able to run on the latest version of OpenJDK.

[4] last update 2019-06-17, Git commit hash `309e1fa`

[5] `https://github.com/biojava/biojava`

avrora	simulates a number of programs run on a grid of AVR microcontrollers
fop	takes an XSL-FO file, parses it and formats it, generating a PDF file.
h2	executes a JDBCbench-like in-memory benchmark, executing a number of transactions against a model of a banking application, replacing the hsqldb benchmark
jython	inteprets the pybench Python benchmark
luindex	Uses lucene to indexes a set of documents; the works of Shakespeare and the King James Bible
lusearch	Uses lucene to do a text search of keywords over a corpus of data comprising the works of Shakespeare and the King James Bible
pmd	analyzes a set of Java classes for a range of source code problems
sunflow	renders a set of images using ray tracing
xalan	transforms XML documents into HTML

Table 4.1: Brief description of selected benchmarks in DaCapo. Copied from the DaCapo manual[6].

[6] http://dacapobench.sourceforge.net/benchmarks.html

4.1.2 SPECjbb2015

SPECjbb2015 is designed to evaluate the performance of Java business applications. The benchmark is bound by CPU, memory and network I/O, but not disk I/O. There is an option to execute the benchmark on more than one machine and then the benchmark would be dependent on network I/O. For our purposes, it is sufficient to measure on a single machine, since the garbage collector is not affected by network I/O. In SPECjbb jargon this means that we run in composite mode (single JVM, single host) [22]. The Java heap size is set to 100GB, as this is the minimum heap size that SPECjbb2015 runs on, due to the pre-committing of memory described in Section 3.6. One benchmark run is designed to take about 2 hours to complete (regardless of what system is used) [22], therefore, there is be no need to simulate long running behavior.

4.1.3 DaCapo Benchmark Configurations

The configurations to be used in simulating (explained in future Section 4.1.5) long running applications can be found in Table 4.2. DaCapo provides up to three sizes of workloads and all of these were utilized whenever possible. There is also an option to configure how many *iterations* of the benchmark that should be executed within one application invocation. Since very short lived application have less time in having the JIT to stabilize the performance, the acceptable stable threshold needs be increased for those configurations. Neither SPECjbb or the synthetic worst-case have any parameters to tune. They are sampled as many times as needed in order to deduce a statistically significant result.

4.1.4 Inferring Confidence Intervals Using the Bootstrap Percentile Method

When a set of observed execution times is collected, calculating a mean value is often of interest. Using only the mean value to deduce conclusion might however give rise to inference-myopia. The observed values may be vastly different, therefore, the mean value might be a poor reflection of data that have high variance. One

Size	Benchmark	Iterations	Use Last Data Points	Stable Threshold	Java Heap
Small	avrora	100	50	3%	4GB
Large	avrora	50	25	3%	4GB
–	biojava	40	25	3%	6GB
Small	fop	1500	100	3%	4GB
Small	h2	100	50	10%	4GB
Large	h2	50	25	3%	4GB
Huge	h2	25	10	3%	6GB
Small	jython	200	100	9%	4GB
Large	jython	50	30	3%	4GB
Small	luindex	100	20	10%	4GB
Small	lusearch	300	30	5%	4GB
Large	lusearch	100	20	3%	4GB
Small	pmd	700	200	10%	4GB
Large	pmd	300	30	10%	4GB
Small	sunflow	700	50	10%	4GB
Large	sunflow	50	10	10%	4GB
Small	xalan	700	100	3%	4GB
Large	xalan	100	20	3%	4GB

Table 4.2: How many data points to use were approximated by the author through a quick sample to see if the resulting data would be likely to have sufficiently low coefficient of variation. The stable threshold is configured to be as low as possible while still resulting in sufficient aggregated data points.

metric that might be even more useful (in conjunction with a mean value) is the 95% confidence interval. This gives an interval where any observed value is within the 95% likelihood.

Since the distribution of the observed execution times is unknown, it makes it more difficult to infer a 95% confidence interval (for the mean). If the underlying data could be assumed to have a Gaussian distribution, then a 95% confidence interval would be given by $[-2\sigma, 2\sigma]$, due to the definition of standard deviation (σ) in a Gaussian distribution.

When the distribution is unknown, a non-parametric method can be used to approximate the confidence interval. One such method is the Percentile Method [23]. Assume $M = m_1, m_2, .., m_k$ number of sets of observations $O_k = o_1, o_2, ..., o_q$ have been recorded. Assume that q and k are not large enough to be able to directly draw an conclusion about the 95% confidence interval. We can then simulate more data (*bootstrapping*) for each set of observation by sampling O_k with replacement x times, which can be referred to as a bootstrap sample $B = b_1, b_2, ..., b_n$. So for an example, b_1 samples O_1 x times, where $x >> q$. This should be performed for each set of observation. Then the 95% confidence interval is inferred by $[p_{2.5}, p_{97.5}]$ on B. This method is used to infer the confidence intervals for all measurements. Each bootstrap sample draws out each observation set 10 000 times, which is assumed to be a large enough data set to draw statistical conclusions from.

4.1.5 Measuring Steady-State Performance in Java Applications using Short Lived Benchmarks

One of Java's great strength is that you can write your code once and then it runs on every system that supports the JVM. The machine can not directly run Java byte code, but have to interpret it and interpreting code is a costly operation, so special measures need to be taken in order to achieve acceptable performance [24].

A common pattern in programming is that the same function would be used over and over again. Java takes advantage of this pattern by implementing a Just-In-Time (JIT) compiler. The JIT compiler compiles and caches code that is frequently executed. A JIT compiler gives a significant performance boost in interpreted languages [24]. So much that, in some cases, the performance of JIT compiled languages can be on par with compiled languages such as C/C++ [25, 26, 27]. However, it takes a while before performance peaks, since the JIT compiler has to compile the code and analyze what parts are executed often enough to be cached. As a result the period after this point is called the *steady-state*. The performance of the steady-state period exhibits less variation in its performance due to most JIT compilation has already been performed. However, the performance during steady-state still has some variability introduced by non-controllable sources such as thread scheduling.

When identifying steady-state there are two issues that need to be resolved. First, long-running applications run typically on large data sets. Benchmarks simulate this by running the same benchmark within one VM invocation multiple times. The first question that needs to be answered is, therefore, how many such iterations are needed to achieve steady-state. Second, different VM invocations may result in different levels of optimizations and therefore has varying steady-state performance [28].

To address these issues, George et al. [28] propose the following guidelines when benchmarking:

1. Consider p VM invocations, each VM invocation running at most q bench-

mark iterations. Suppose that we want to retain k measurements per invocation.

2. For each VM invocation, i, determine the iterations s_i where steady-state performance is reached, *i.e.,* once the coefficient of variation of the k iterations ($s_i - k$ to s_i) falls below a preset threshold, say 0.01 or 0.02

3. For each VM invocation, compute the mean $\overline{x_i}$ of the k benchmark iteration under steady-state:

$$\overline{x_i} = \sum_{j=s_i-k}^{s_i} x_{ij}$$

4. Compute the confidence interval for a given confidence level across the computed means from the different VM invocations. The overall mean equals $\overline{x} = \sum_{i=1}^{p} \overline{x_i}$ and the confidence interval is computed over the $\overline{x_i}$ measurements.

Step 2 approximates where the JIT compiler performance boosting have stabilized. We are only interested in measuring differences in the garbage collector algorithm. Finding the point where the JIT compiler have done most of its work is, therefore, key in order to be able to compare results. Results might still have high variance, due to factors we cannot control, such as thread scheduling from the operating system, therefore, we should disregard any data point that have far too great of coefficient of variation. In step 3–4 the selected data is aggregated in order to be able to find a mean and a confidence interval.

4.2 Measuring Reallocation Work

To understand other impacts from the new design, we run a separate batch of benchmarks with logging is enabled [7]. It should be noted that no additional metrics have been inserted into the source code. The same machine used to measure throughput is also used to measure reallocation work. Since logging might effect performance data, it is important to not collect logging information at the same time as performance measurements. From the log data, we can see the number of garbage collection cycles that is performed and how much data that is moved during the execution of the program.

[7] This is enabled by supplying `-Xlog:gc+reloc` to the JVM during startup.

4.3 Machines to Collect Data

Two machines are used to collect data simultaneously, machine A for the DaCapo site and machine B for SPECjbb2015. The specification for each machine is found in Table 4.3. OpenJDK is compiled using GCC 7.5.0 and the patch containing the new implemenation is applied on top of commit from the release branch of OpenJDK with id eccdd8e60399a4b0db77b94e77bb32381090a5c6[8].

[8] authored on 2020-02-12

Machine	Type	OS	Linux Kernel Version	CPU	CPUs	Threads Per Core	Core Per Sockets	CPU Attached To Number Of Sockets	Memory
A	Native	Ubuntu 18.04.4 LTS	4.15.0-99-generic	Intel(R) Xeon(R) CPU E5-2665 0 @ 2.40GHz	32	2	8	2	32G
B	Virtual Machine (VMWare)	Ubuntu 18.04.3 LTS	4.15.0-96-generic	Intel(R) Xeon(R) CPU E5-2697 v2 @ 2.70GHz	8	1	2	4	128G

Table 4.3: Machines used to collect data.

Chapter 5

Results

5.1 SPECjbb2015

SPECjbb2015 has two throughput measurements, critical-jOPS and max-jOPS which measures benchmark throughput in two distinct ways. Critical-jOPS measures throughput under a response time constraint and max-jOPS measures maximal throughput [22]. In Table 5.1 the results are presented and visualized in Figure A.1, which shows that there is a significant difference between the new design and ZGC. Both max-jOPS and critical-jOPS have a performance regression of 2% for the new design.

Type	N	Mean (jOPS)	Standard Deviation	Relative Standard Deviation	Performance	CI Lower	CI Upper
vanilla	91	9044	128	1.41%	0.00%	8991	9096
compact	91	8883	133	1.50%	-1.78%	8829	8938

(a) max-jOPS

Type	N	Mean (jOPS)	Standard Deviation	Relative Standard Deviation	Performance	95-CI Lower	95-CI Upper
vanilla	91	5509	76	1.37%	0.00%	5477	5539
compact	91	5385	72	1.33%	-2.25%	5354	5414

(b) critical-jOPS

Table 5.1: SPECjbb2015 results.
A performance regression of 2% is seen for both max-jOPS and critical-jOPS.

5.2 DaCapo

The DaCapo results are presented in Table 5.2 and visualized in Figures A.3–A.12. The results can be grouped into three categories: (1) no difference in execution time performance[1], (2) performance regression in execution time for the new solution, (3) performance improvement in execution time for the new solution. No difference (1) in performance was observed in the following benchmarks: avrora_large, biojava, h2_small, h2_large, luindex_small, lusearch_small, jython_small, pmd_small, xalan_small, xalan_large. Performance regression in execution time for the new solution was observed in benchmarks: avrora_small, h2_huge, lusearch_large, jython_large, pmd_large, sunflow_small, in the range of 0.93–3.76%. Performance improvements (3) in execution time was observed in fop_small, sunflow_large and

[1] Only results that have non-overlapping confidence interval should be considered to have statistically significant difference between means.

speed up was measured to 5.69%, respectively 22.42%.

Benchmark	Type	N	Mean (ms)	Standard Deviation	Relative Standard Deviation	Performance	CI Lower	CI Upper
avrora_small	vanilla	47	2021	12	0.61%	0.00%	2013	2028
avrora_small	compact	88	2039	16	0.77%	-0.93%	2031	2049
avrora_large	vanilla	30	111106	1227	1.10%	0.00%	110296	111648
avrora_large	compact	30	110993	2014	1.81%	0.10%	109614	111842
biojava	vanilla	31	28884	875	3.03%	0.00%	28363	29363
biojava	compact	32	28956	846	2.92%	-0.25%	28435	29407
fop_small	vanilla	34	56	1	1.79%	0.00%	55	56
fop_small	compact	50	53	1	1.68%	5.69%	52	53
h2_small	vanilla	45	1118	17	1.56%	0.00%	1107	1128
h2_small	compact	50	1098	20	1.80%	1.77%	1087	1109
h2_large	vanilla	23	65436	1048	1.60%	0.00%	64783	65991
h2_large	compact	23	65737	1069	1.63%	-0.46%	65143	66369
h2_huge	vanilla	20	580297	9441	1.63%	0.00%	574809	585738
h2_huge	compact	20	592139	9772	1.65%	-2.04%	586628	597638
luindex_small	vanilla	24	115	4	3.26%	0.00%	112	117
luindex_small	compact	25	114	4	3.47%	0.53%	112	116
lusearch_small	vanilla	13	72	3	3.66%	0.00%	70	73
lusearch_small	compact	25	75	3	4.45%	-3.70%	72	77
lusearch_large	vanilla	13	389	9	2.43%	0.00%	383	394
lusearch_large	compact	25	403	9	2.35%	-3.76%	395	407
jython_small	vanilla	20	218	6	2.82%	0.00%	214	221
jython_small	compact	50	219	6	2.73%	-0.43%	215	223
jython_large	vanilla	21	12890	136	1.06%	0.00%	12811	12968
jython_large	compact	21	13109	121	0.92%	-1.70%	13039	13179
pmd_small	vanilla	37	19	0	0.98%	0.00%	19	19
pmd_small	compact	50	19	1	3.18%	0.82%	18	19
pmd_large	vanilla	50	2739	29	1.04%	0.00%	2722	2755
pmd_large	compact	50	2801	25	0.91%	-2.24%	2786	2815
sunflow_small	vanilla	7	344	11	3.18%	0.00%	338	350
sunflow_small	compact	50	360	6	1.64%	-4.69%	354	362
sunflow_large	vanilla	50	1984	35	1.76%	0.00%	1964	2005
sunflow_large	compact	50	1539	13	0.83%	22.42%	1532	1547
xalan_small	vanilla	27	56	1	1.57%	0.00%	55	56
xalan_small	compact	50	57	2	3.17%	-1.36%	55	57
xalan_large	vanilla	42	2884	75	2.61%	0.00%	2840	2928
xalan_large	compact	50	2839	88	3.11%	1.56%	2794	2887

Table 5.2: DaCapo results.
No difference in performance was observed in: {avrora_large, biojava, h2_small, h2_large, luindex_small, lusearch_small, jython_small, pmd_small, xalan_small, xalan_large}. Performance regression in execution time for: {avrora_small, h2_huge, lusearch_large, jython_large, pmd_large, sunflow_small}, in the range of 0.93–3.76%. Performance improvements in execution time in: {fop_small, sunflow_large}.

5.3 Behavioral Impact

Tables 5.3–5.5 show that for a majority of the benchmarks less data (in MB) is relocated using the new design. This was observed in the following benchmarks: biojava, fop_small, h2_small, h2_large, h2_huge, jython_large, luindex_small, lusearch_small, luindex_large, pmd_small, pmd_large, sunflow_small, xalan_small, and xalan_large. In jython_small and sunflow_large the opposite was observed, where more MB is relocated for the new design, furthermore in avrora_small and avrora_large no difference in amount of relocated MB was observed. Table 5.3–Table 5.5 show that

a majority of the benchmarks have no difference in the amount of garbage collection cycles. This was observed in the following benchmarks: avrora_small, avrora_large, biojava, fop_small, h2_small, jython_small, jython_large, luindex_small, lusearch_small, luindex_large, pmd_small, pmd_large, sunflow_small, xalan_small, and xalan_large. In h2_large and h2_huge we can observe that fewer garbage collection cycles is performed with the new design and in sunflow_large more cycles is performed with the new design.

Measurement	Type	N	Mean	Standard Deviation	Relative Standard Deviation	CI Lower	CI Upper
Sum Relocated MB	vanilla	199	75405 (MB)	4045	5.36%	73098	77816
Sum Relocated MB	compact	199	72902 (MB)	3688	5.06%	70825	75113
GC Rounds	vanilla	199	103 (rounds)	5	5.31%	99	105
GC Rounds	compact	199	103 (rounds)	5	4.97%	99	105

Table 5.3: Results of measuring garbage collection work in SPECjbb2015. While the mean of sum relocated MB is lower for the new design, the confidence intervals overlap so no statistical conclusion can be drawn.

Benchmark	Type	Measurement	N	Mean	Standard Deviation	Relative Standard Deviation	CI Lower	CI Upper
avrora_small	Sum Relocated MB	vanilla	60	203 (MB)	26	12.57%	200	218
avrora_small	Sum Relocated MB	compact	60	200 (MB)	0	0.00%	200	200
avrora_small	GC Cycles	compact	60	100 (cycles)	0	0.00%	100	100
avrora_small	GC Cycles	compact	60	100 (cycles)	0	0.00%	100	100
avrora_large	Sum Relocated MB	vanilla	60	103 (MB)	18	17.17%	100	117
avrora_large	Sum Relocated MB	compact	60	100 (MB)	0	0.00%	100	100
avrora_large	GC Cycles	vanilla	60	50 (cycles)	0	0.00%	50	50
avrora_large	GC Cycles	compact	60	50 (cycles)	0	0.00%	50	50
biojava	Sum Relocated MB	vanilla	60	14176 (MB)	46	0.32%	14148	14201
biojava	Sum Relocated MB	compact	60	13729 (MB)	31	0.23%	13711	13746
biojava	GC Cycles	vanilla	60	202 (cycles)	0	0.00%	202	202
biojava	GC Cycles	compact	60	202 (cycles)	0	0.13%	202	202
fop_small	Sum Relocated MB	vanilla	60	11098 (MB)	920	8.29%	10518	11580
fop_small	Sum Relocated MB	compact	60	3006 (MB)	1	0.02%	3005	3006
fop_small	GC Cycles	vanilla	60	1500 (cycles)	0	0.00%	1500	1500
fop_small	GC Cycles	compact	60	1500 (cycles)	0	0.00%	1500	1500
h2_small	Sum Relocated MB	vanilla	60	1253 (MB)	105	8.36%	1189	1314
h2_small	Sum Relocated MB	compact	60	336 (MB)	18	5.45%	325	347
h2_small	GC Cycles	vanilla	60	101 (cycles)	0	0.00%	101	101
h2_small	GC Cycles	compact	60	101 (cycles)	0	0.00%	101	101
h2_large	Sum Relocated MB	vanilla	60	6831 (MB)	333	4.87%	6631	7020
h2_large	Sum Relocated MB	compact	60	5439 (MB)	283	5.20%	5273	5601
h2_large	GC Cycles	vanilla	60	166 (cycles)	11	6.79%	159	172
h2_large	GC Cycles	compact	60	130 (cycles)	9	7.23%	124	135
h2_huge	Sum Relocated MB	vanilla	40	27793 (MB)	593	2.13%	27436	28125
h2_huge	Sum Relocated MB	compact	40	26289 (MB)	478	1.82%	26017	26576
h2_huge	GC Cycles	vanilla	40	239 (cycles)	11	4.60%	232	244
h2_huge	GC Cycles	compact	40	221 (cycles)	10	4.50%	215	226
jython_small	Sum Relocated MB	vanilla	60	4006 (MB)	0	0.01%	4005	4006
jython_small	Sum Relocated MB	compact	60	6774 (MB)	0	0.00%	6774	6774
jython_small	GC Cycles	vanilla	60	201 (cycles)	0	0.00%	201	201
jython_small	GC Cycles	compact	60	201 (cycles)	0	0.00%	201	201
jython_large	Sum Relocated MB	vanilla	60	2250 (MB)	177	7.88%	2156	2362
jython_large	Sum Relocated MB	compact	60	2121 (MB)	22	1.02%	2108	2133
jython_large	GC Cycles	vanilla	60	227 (cycles)	11	4.69%	221	233
jython_large	GC Cycles	compact	60	235 (cycles)	13	5.36%	227	242
luindex_small	Sum Relocated MB	vanilla	60	398 (MB)	0	0.00%	398	398
luindex_small	Sum Relocated MB	compact	60	200 (MB)	1	0.28%	200	200
luindex_small	GC Cycles	vanilla	60	100 (cycles)	0	0.00%	100	100
luindex_small	GC Cycles	compact	60	100 (cycles)	0	0.00%	100	100
lusearch_small	Sum Relocated MB	vanilla	60	891 (MB)	298	33.48%	712	1087
lusearch_small	Sum Relocated MB	compact	60	600 (MB)	0	0.00%	600	600
lusearch_small	GC Cycles	vanilla	60	300 (cycles)	0	0.00%	300	300
lusearch_small	GC Cycles	compact	60	300 (cycles)	0	0.00%	300	300
lusearch_large	Sum Relocated MB	vanilla	60	1700 (MB)	48	2.83%	1670	1725
lusearch_large	Sum Relocated MB	compact	60	1379 (MB)	26	1.92%	1362	1393
lusearch_large	GC Cycles	vanilla	60	203 (cycles)	1	0.70%	202	204
lusearch_large	GC Cycles	compact	60	203 (cycles)	1	0.72%	202	203

Table 5.4: Results of measuring garbage collection work in the DaCapo suite (a).

Benchmark	Type	Measurement	N	Mean	Standard Deviation	Relative Standard Deviation	CI Lower	CI Upper
pmd_small	Sum Relocated MB	vanilla	60	2798 (MB)	0	0.00%	2798	2798
pmd_small	Sum Relocated MB	compact	60	1402 (MB)	0	0.02%	1401	1402
pmd_small	GC Cycles	vanilla	60	700 (cycles)	0	0.00%	700	700
pmd_small	GC Cycles	compact	60	700 (cycles)	0	0.00%	700	700
pmd_large	Sum Relocated MB	vanilla	60	880 (MB)	11	1.28%	872	883
pmd_large	Sum Relocated MB	compact	60	491 (MB)	5	1.08%	487	493
pmd_large	GC Cycles	vanilla	60	102 (cycles)	0	0.00%	102	102
pmd_large	GC Cycles	compact	60	102 (cycles)	0	0.00%	102	102
sunflow_small	Sum Relocated MB	vanilla	60	1200 (MB)	0	0.00%	1200	1200
sunflow_small	Sum Relocated MB	compact	60	602 (MB)	0	0.00%	602	602
sunflow_small	GC Cycles	vanilla	60	300 (cycles)	0	0.00%	300	300
sunflow_small	GC Cycles	compact	60	300 (cycles)	0	0.00%	300	300
sunflow_large	Sum Relocated MB	vanilla	60	2956 (MB)	135	4.58%	2876	3039
sunflow_large	Sum Relocated MB	compact	60	3403 (MB)	79	2.32%	3357	3449
sunflow_large	GC Cycles	vanilla	60	197 (cycles)	5	2.48%	194	199
sunflow_large	GC Cycles	compact	60	248 (cycles)	6	2.24%	244	251
xalan_small	Sum Relocated MB	vanilla	60	2777 (MB)	175	6.29%	2676	2800
xalan_small	Sum Relocated MB	compact	60	1402 (MB)	0	0.00%	1402	1402
xalan_small	GC Cycles	vanilla	60	700 (cycles)	0	0.00%	700	700
xalan_small	GC Cycles	compact	60	700 (cycles)	0	0.00%	700	700
xalan_large	Sum Relocated MB	vanilla	60	6751 (MB)	55	0.82%	6717	6783
xalan_large	Sum Relocated MB	compact	60	5845 (MB)	23	0.40%	5831	5858
xalan_large	GC Cycles	vanilla	60	503 (cycles)	1	0.24%	502	504
xalan_large	GC Cycles	compact	60	503 (cycles)	1	0.24%	502	504

Table 5.5: Results of measuring garbage collection work in the DaCapo suite (b).

Chapter 6

Conclusions

Moving garbage collectors that perform concurrent relocation, need to store forwarding information. In ZGC, the memory overhead due to forwarding information may become as large as the application heap itself. In other collectors, like Shenandoah and ART, the way forwarding information is stored delays the recycling of old pages. This thesis shows it is possible to design and implement a compressed forwarding table with a guaranteed upper limit of < 3.2% memory overhead for storing forwarding information. This design adds an additional computational cost for lookups of forwarding addresses. Several benchmarks were used to understand the effects of the compact forwarding table design on application throughput. Evaluating the computational impact, three groups of statistically significant results is found: performance regression in the range of 1–3% (SPECjbb2015 and 6 DaCapo configurations), no difference (10 DaCapo configurations), performance improvements (2 DaCapo configurations). The goal of a maximum 5% performance regression is thus considered to be fulfilled.

The new implementation requires 961 lines of code, which is only 68 more lines compared to the original solution. While lines of code is generally not a good metric of code quality, it is still something worth to be considered in a large code base such as OpenJDK.

As future work, the execution time can be improved by optimizing the scanning of the live map. The current implementation will scan for liveness bit by bit in a loop. Most of the bits in the live map will consist of zeros. There exists a special machine instruction that will count the number of zeroes [29] and utilizing this machine instruction should, therefore, be able to significantly reduce the computational cost, since most iterations in the scanning of the live map can be eliminated.

There is one major shortcut in the current implementation which hinders its adoption into OpenJDK. As mentioned in mentioned in Section 3.6, all memory that is needed for relocation is committed already at the setup of the compact forwarding tables. This pre-committing of memory leads to a need for much larger heaps than with the original solution. This is done because the authored focused on having a correct prototype implementation completed in time and this simplification saved a lot of time. Postponing committing memory until the relocation actually takes places is supported by the design presented in this thesis.

At first glance, the design should be applicable in other garbage collectors than ZGC, like Shenandoah and ART. Inspecting Shenandoah shows that it could adopt the forwarding table with some additional changes. Currently Shenandoah marks regions of virtual memory to be relocated. Such a region cannot be free'd until

every object in that region is relocated and all pointers are updated to the new location. Shenandoah stores the new address in the old objects in the marked region, meaning that they have the worst-case scenario of ZGC (100% memory overhead) with regards to memory overhead per page.

Even though adopting a forwarding table would change Shenandoah so that forward addresses are not stored in old relocated objects, it would still not allow freeing the memory, since the virtual address is used to signify that this object have been relocated. To get any memory benefits, virtual and physical memory needs to be decoupled. As soon as an object have been relocated the physical memory could be reused, but the virtual memory may not be reused until next garbage collection cycle. Decoupling physical memory from virtual memory in a garbage collector has been done before, notably in C4 [30]. In order to achieve good performance with decoupled virtual and physical memory C4 had to create a custom Linux kernel. Substantial work may therefore have to been done for Shenandoah to be able to adopt this design, but the performance implications is not clear.

Another future work would be to investigate the differences in performance between our implementation and Compressor. As mentioned previously in Chapter 2, Compressor is a garbage collector that on a high level has design that resembles our with regards to calculating forwarding addresses.

Bibliography

[1] Hosking A, Jones R, Moss E. The Garbage Collection Handbook. Chapman and Hall/CRC; 2016.

[2] Oracle Corporation. openjdk/jdk: Read-only mirror of https://hg.openjdk.java.net/jdk/jdk (commit: eccdd8e60399a4b0db77b94e77bb32381090a5c6);. Available from: https://github.com/openjdk/jdk.

[3] Dijkstra EW, Lamport L, Martin AJ, Scholten CS, Steffens EF. On-the-fly garbage collection: An exercise in cooperation. Communications of the ACM. 1978;21(11):966–975.

[4] Silberschatz A, Galvin PB, Gagne G. Operating System Concepts. 9th ed. Wiley Publishing; 2012.

[5] Rovner P. On Adding Garbage Collection and Runtime Types to a Strongly-Typed, Statically-Checked, Concurrent Language. Xerox Corporation, Palo Alto Research Center; 1985.

[6] Mobile Operating System Market Share Worldwide;. Available from: https://gs.statcounter.com/os-market-share/mobile/worldwide.

[7] Android 8.0 ART Improvements : Android Open Source Project;. Available from: https://source.android.com/devices/tech/dalvik/improvements.

[8] Android 10 for Developers : Android Developers;. Available from: https://developer.android.com/about/versions/10/highlights.

[9] Oracle JDK vs OpenJDK and Java JDK Development Process;. Available from: https://javapapers.com/java/oracle-jdk-vs-openjdk-and-java-jdk-development-process/.

[10] How to contribute;. Available from: https://openjdk.java.net/contribute/.

[11] Types of Java Garbage Collectors;. Available from: https://javapapers.com/java/types-of-java-garbage-collectors.

[12] Schatzl T. JEP 363: Remove the Concurrent Mark Sweep (CMS) Garbage Collector; 2019. Accessed: 2019-04-22. https://openjdk.java.net/jeps/363.

[13] Lidén P, Karlsson S. JEP 333: ZGC: A Scalable Low-Latency Garbage Collector; 2018. Accessed: 2019-04-05. http://openjdk.java.net/jeps/333.

[14] Flood CH, Kennke R, Dinn A, Haley A, Westrelin R. Shenandoah: An open-source concurrent compacting garbage collector for OpenJDK. In: Proceedings of the 13th International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools; 2016. p. 1–9.

[15] Yang AM, Österlund E, Wrigstad T. Improving Program Locality in the GC Using Hotness. In: Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation. PLDI 2020. New York, NY, USA: Association for Computing Machinery; 2020. p. 301–313. Available from: https://doi.org/10.1145/3385412.3385977.

[16] Kocher P, Horn J, Fogh A, Genkin D, Gruss D, Haas W, et al. Spectre attacks: Exploiting speculative execution. In: 2019 IEEE Symposium on Security and Privacy (SP). IEEE; 2019. p. 1–19.

[17] Gabbay F, Mendelson A. Speculative execution based on value prediction. Citeseer; 1996.

[18] Yang AM, Österlund E, Wilhelmsson J, Nyblom H, Wrigstad T. ThinGC: Complete Isolation with Marginal Overhead. In: Proceedings of the 2020 ACM SIGPLAN International Symposium on Memory Management. ISMM 2020. New York, NY, USA: Association for Computing Machinery; 2020. p. 74–86. Available from: https://doi.org/10.1145/3381898.3397213.

[19] Kermany H, Petrank E. The Compressor: concurrent, incremental, and parallel compaction. In: Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation; 2006. p. 354–363.

[20] Cormen TH, Leiserson CE, Rivest RL, Stein C. Introduction to algorithms. MIT press; 2009.

[21] Blackburn SM, Garner R, Hoffman C, Khan AM, McKinley KS, Bentzur R, et al. 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. New York, NY, USA: ACM Press; 2006. p. 169–190.

[22] Standard Performance Evaluation Corporation. SPECjbb2015 Benchmark User Guide; 2017. Available from: https://www.spec.org/jbb2015/docs/userguide.pdf.

[23] Efron B, Hastie T. Computer age statistical inference. vol. 5. Cambridge University Press; 2016.

[24] Schildt H. Java: A Beginner's Guide, Sixth Edition. Beginner's Guide. McGraw-Hill Education; 2014.

[25] Sestoft P. Numeric performance in C, C# and Java. IT University of Copenhagen, Denmark. 2010;185:186.

[26] Costanza P, Herzeel C, Verachtert W. A comparison of three programming languages for a full-fledged next-generation sequencing tool. BMC bioinformatics. 2019;20(1):301.

[27] Källén M, Wrigstad T. Performance of an OO compute kernel on the JVM: revisiting Java as a language for scientific computing applications. In: Proceedings of the 16th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes; 2019. p. 144–156.

[28] Georges A, Buytaert D, Eeckhout L. Statistically Rigorous Java Performance Evaluation. ACM SIGPLAN Notices. 2007;42(10):57–76.

[29] Intel. Intel® 64 and IA-32 Architectures Software Developer's Manual Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D, and 4. 2018;.

[30] Tene G, Iyengar B, Wolf M. C4: The continuously concurrent compacting collector. In: Proceedings of the international symposium on Memory management; 2011. p. 79–88.

Glossary

collector "a system component responsible for garbage collection" 1.

dangling pointer a pointer to an object, which memory has been returned to the memory manager 1.

mutator the user program, which mutates the objects to be collected by the garabge collector 1.

root "a reference that is directly accessible to the mutator without going through other objects" 1.

Appendix A

Plots

A.1 SPECjbb2015 Plots

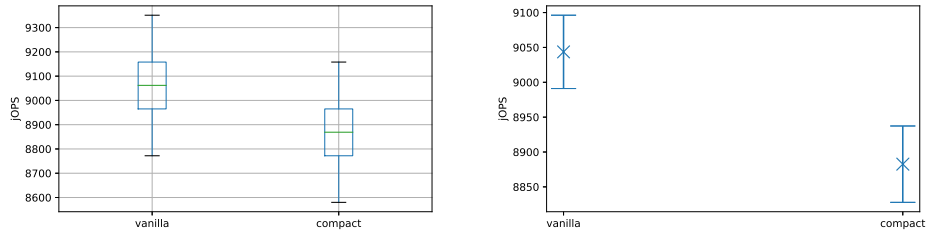


Figure A.1: A performance regression of 2% is seen for the new design

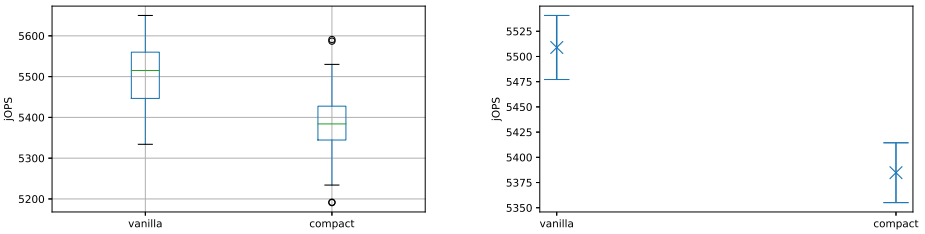
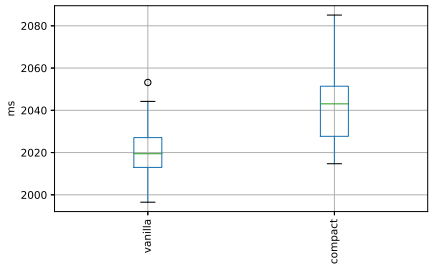
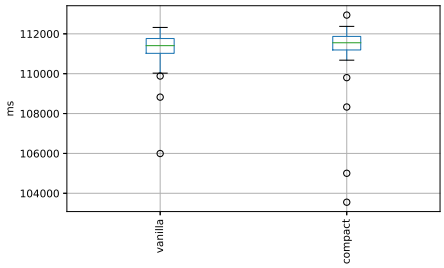


Figure A.2: A performance regression of 2% is seen for the new design

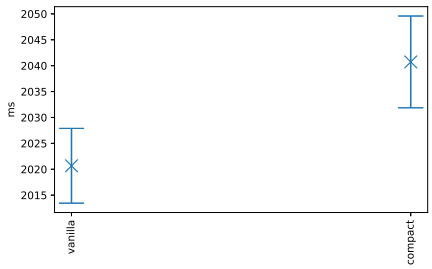
A.2 DaCapo Plots



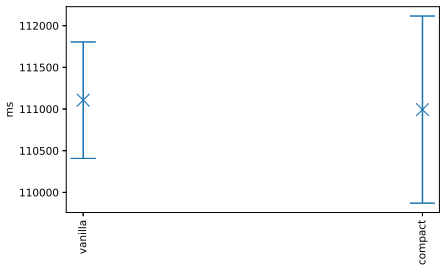
(a) avrora small



(b) avrora large

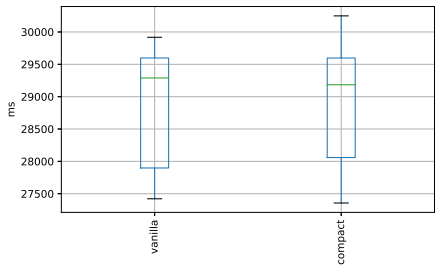


(c) avrora small

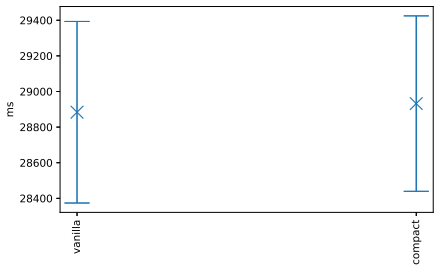


(d) avrora large

Figure A.3: avrora results.
A performance regression in avrora small and no statistically significant difference in avrora large since overlapping confidence intervals.

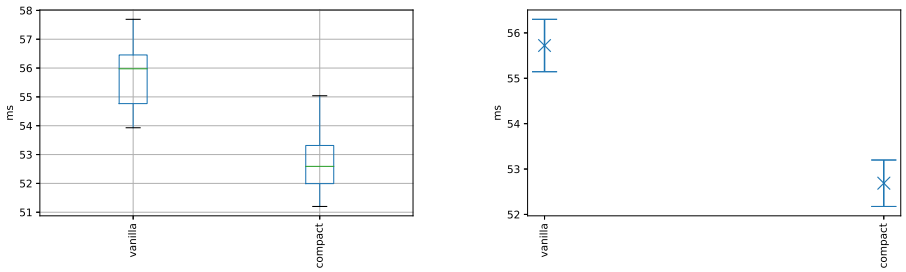


(a) biojava



(b) biojava

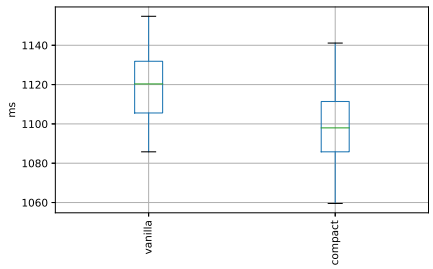
Figure A.4: biojava results.
No statistically significant difference in biojava since overlapping confidence intervals.



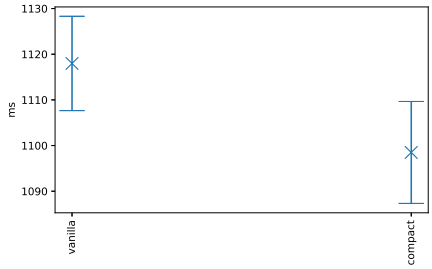
(a) fop small

(b) fop small

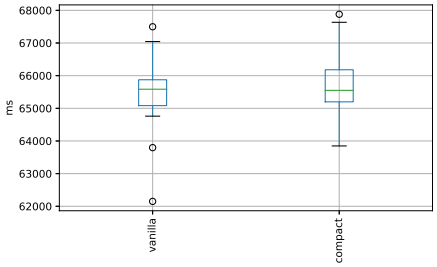
Figure A.5: fop results.
Performance improvement for the new design.



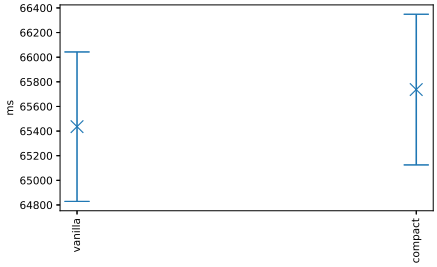
(a) h2 small



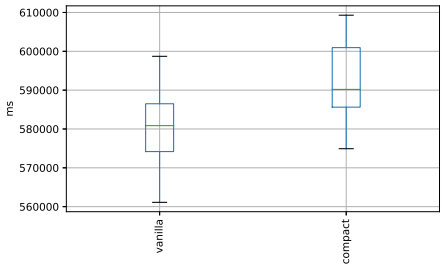
(b) h2 small



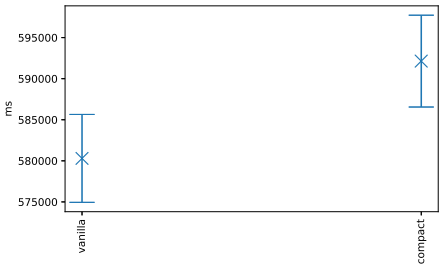
(c) h2 large



(d) h2 large

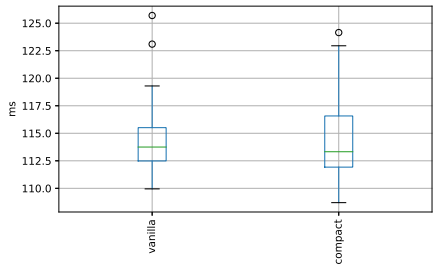


(e) h2 huge

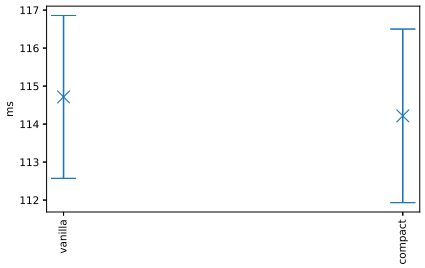


(f) h2 huge

Figure A.6: h2 results.
No statistically significant difference can be observed in h2 small and large since overlapping confidence intervals. A performance regression is seen in h2 huge.

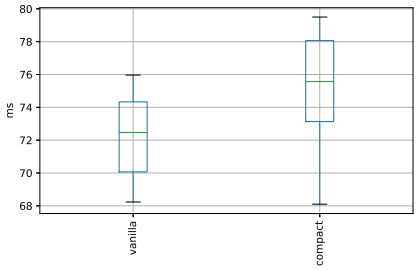


(a) luindex small

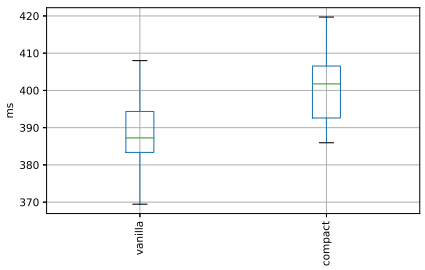


(b) luindex small

Figure A.7: luindex results.
No statistically significant difference since overlapping confidence intervals.



(a) lusearch small

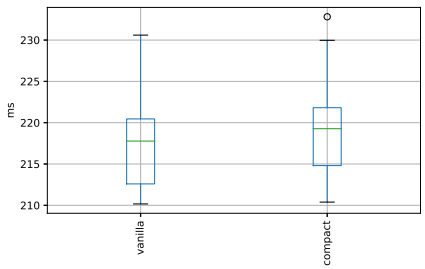


(b) lusearch large

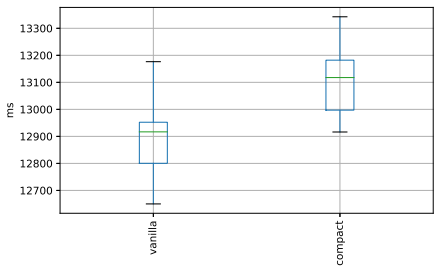
(c) lusearch small

(d) lusearch large

Figure A.8: lusearch results.
No statistically significant difference in lusearch small since overlapping confidence intervals. A performance regression is seen in lusearch large.

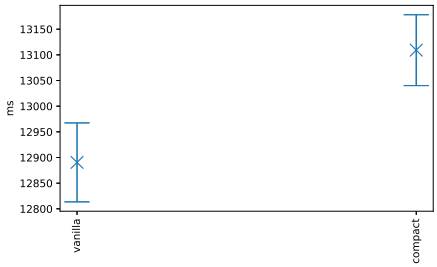


(a) jython small



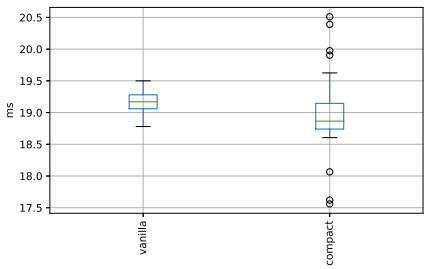
(b) jython large

(c) jython small

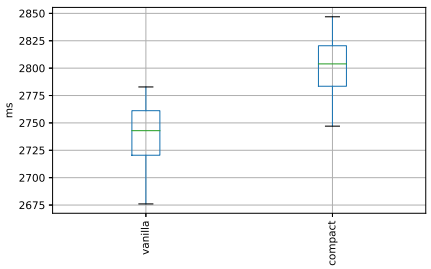


(d) jython large

Figure A.9: jython results.
No statistically significant difference in jython small since overlapping confidence intervals. A performance regression is seen in jython large.

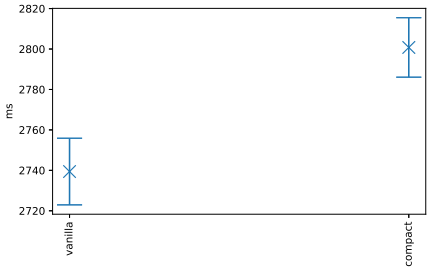


(a) pmd small



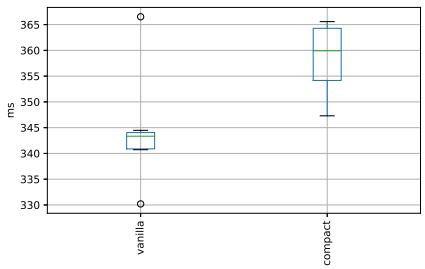
(b) pmd large

(c) pmd small

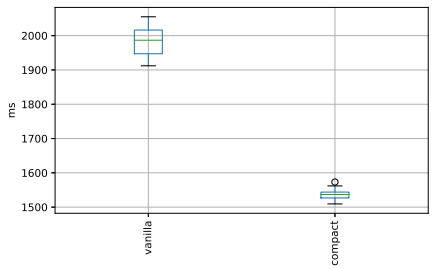


(d) pmd large

Figure A.10: pmd results.
No statistically significant difference in pmd small since overlapping confidence intervals. A performance regression is seen in pmd large.

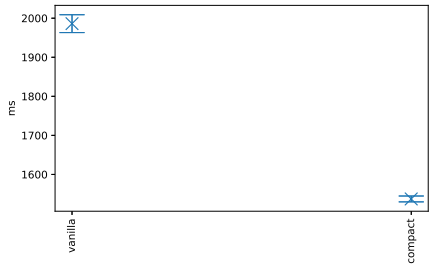


(a) sunflow small



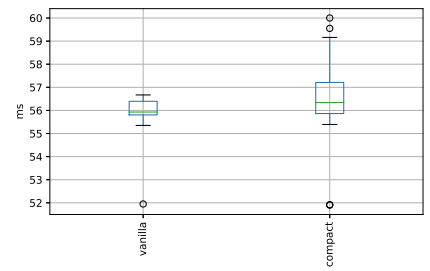
(b) sunflow large

(c) sunflow small

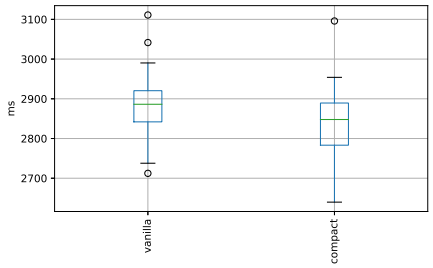


(d) sunflow large

Figure A.11: pmd results.
A performance regression is seen in sunflow small and a performance improvement is seen in sunflow large.

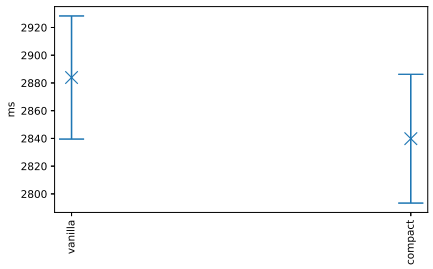


(a) xalan small



(b) xalan large

(c) xalan small



(d) xalan large

Figure A.12: xalan results.
No statistically significant difference in xalan small and large since overlapping confidence intervals.