UPPSALA
UNIVERSITET

UPTEC IT 23011
Examensarbete 300 hp
Juni 2023

Compressing Pointers for the Z Garbage Collector

Runtime compression of pointers in a concurrent setting

Linus Shoravi

Civilingenjörsprogrammet i informationsteknologi

UPPSALA
UNIVERSITET

Compressing Pointers for the Z Garbage Collector

Linus Shoravi

Abstract

Pointers in 64-bit architectures are unlikely to exhaust their vast address range, and are as such needlessly big. Reducing the amount of memory a pointer occupies leads to reduced memory demands, better usage of memory, and better locality. Pointer compression is a term that encompasses techniques that aim to make pointers occupy less memory, often to 32-bit for the sake of word alignment. Pointers that are 32-bit embody the opposite problem of having too restricted of an address range, being able to address only 4 GB. Z is a garbage collector in the HotSpot JVM which does not support pointer compression. Partly because the aforementioned address range restriction, and partly because the implementation of compressed pointers which exist in HotSpot would clash with the goals of the garbage collector. This project explores ways of implementing pointer compression for Z that isn't detrimental to the goals of the garbage collector, and aims to find where problems may occur. The outset was to explore compressing speculatively during runtime. The result is a design that relies on a custom bit layout for compressed pointers, inspecting bit layouts of the pointers on each read and write to detect the compression status. This seems to be the most promising in terms of code maintainability and ease of implementation.

Teknisk-naturvetenskapliga fakulteten

Uppsala universitet, Utgivningsort Uppsala

Handledare: Erik Österlund Ämnesgranskare: Tobias Wrigstad

Examinator: Lars-Åke Nordén

Sammanfattning

En *pekare* är i ett datorprogram ett värde som avser en plats i datorns minne. På moderna datorer så ockuperar pekare åtta bytes, men ingen dator idag utnyttjar alla åtta bytes som address, pekare är alltså onödigt stora. Pekarkomprimering är en term som beskriver tekniker vilka avser att minska mängden minne en pekare utrymmer. Pekare brukar komprimeras till 4 bytes. Pekare som är 4 bytes breda har problemet att de har en begränsad addressrymd av endast 4 GB. Z är en *skräpsamlare* som inte stöder pekarkomprimering av just denna begränsning. En skräpsamlare är ett verktyg i ett program som hjälper programmeraren att hantera datorns minne. Z har även hårda krav på prestanda, eftersom att den är avsedd för program som vill ha kort svarstid, exempelvis webbsidor. Detta examensarbete utforskar sätt att implementera pekarkomprimering för Z så att dess mål fortfarande uppnås, och hitta vart problemen uppstår. Utgångspunkten var att utforska att göra pekarkomprimering under körningstid. Resultatet är en design för pekarkomprimering för Z. Denna design verkar mest lovande i termer av underhållbarhet och lätthet att implementera. Designen har vissa hittils olösta problem till vilka det finns en föreslagen lösning. Rapporten detaljerar även vilka problem som kvarstår att lösa.

Acknowledgement Although an individual project, this thesis is not the work of the author alone. I would like to extend my sincere gratitude towards my supervisor Erik Österlund for his fantastic help, Jonas Norlinder for providing me with a launchpad, my reviewer Tobias Wrigstad for his trust in me and this project, and the people at Oracle for fostering a great and welcoming environment in which to experiment. Finally, a thank you and a sorry to my friends, family and partner, who occasionally had to suffer through my boring, long-winded and frustration-fueled whining.

Contents

List of Terms & Abbreviations

Pointer A value which represents a place in computer memory.

Pointer compression Technique which aims to make pointers occupy less memory.

JVM Java Virtual Machine, a virtual execution environment for Java programs.

HotSpot A JVM developed by Oracle.

GC Garbage Collector, provides automatic memory management for programs.

ZGC The Z garbage collector, a concurrent garbage collector found in the HotSpot JVM.

Mutator thread Computer threads that execute user (Java) application code.

GC thread Computer threads that execute garbage collection code.

JIT compilation Just-in-time compilation, compiling to machine code during runtime.

1 Introduction

Pointers in 64-bit architectures are unlikely to exhaust their vast address range, and are as such needlessly big. Pointer compression is a concept that encompasses techniques which aim to make pointers occupy less memory, often to 32-bit for the sake of word alignment. Pointers that are 32-bit embody the opposite problem of having too restricted address range, limited to only 4 GB. Z is a garbage collector found in the HotSpot JVM which does not support pointer compression. Partly because of the lack of address range, and partly because the implementation of compressed pointers which exist in HotSpot would clash with the intended use case of the garbage collector.

1.1 Purpose and Goals

The goal of this project is to explore ways of implementing pointer compression for Z that are not detrimental to the goals of the garbage collector. The current implementation of compressing pointers in HotSpot makes the decision of compressing at startup, and compresses all pointers. The starting point for this project was to perform compression *speculatively*, and only on pointers belonging to certain classes. Hence, this implementation suggests taking the decision *during runtime* on a *class-level*, as opposed to at all pointers at startup. Making this decision during runtime implies changing objects on-the-fly.

1.2 Delimitations

The main limitation of this project is its scope: about 20 person weeks simply is not enough to implement this feature fully, which was well-known from the start. In particular, supporting JIT-compilation as well as dynamically determining what classes pointers should be compressed and to what extent is left as future work, and only user-defined classes which no other class inherits from have been tested. This is a reasonable approach: solve the simplest possible case, and then generalise and extend.

1.3 Summary of Results

Implementing dynamic pointer compression in ZGC proved difficult on several accounts, mostly in relation to concurrency: changing an object on-the-fly requires that all places in the code that refer to this object change their view accordingly, or the result of a field read or write is likely corrupt data. This work resulted in a prototype implementation that relies on a different representation for compressed and uncompressed pointers and inspecting the pointer each read and write to detect the layout. This design seems to be the most promising when compared to the other attempts described in Section 6.4, in terms of code maintainability as well as ease of implementation. It circumvents the need to refactor large parts of the code base or maintain several hand-written, almost-same-but-different assembly code paths that one of the other attempted implementations suffered from, detailed in

Section 6.4.

2 Background

This section covers the necessary background information needed to understand the problem and the method for exploring it. The goal of the project is ultimately to implement runtime pointer compression of so-called *ordinary object pointers*, but there is a great deal of context around it which is needed to understand why it's being done, and how that would look. This section hence is a high-level description of what ordinary object pointers are, what pointer compression is, and why pointer compression is desirable. What follows is a short description of HotSpot and the current implementation of compressed pointers in HotSpot, the Z garbage collector and its goals, and why the standard implementation of pointer compression in OpenJDK is not applicable to ZGC. Finally, some core concepts and implementation details regarding objects in HotSpot are explained.

We start by covering some concepts necessary for the understanding of the rest of the thesis.

2.1 Memory Management

Certain systems languages such as C require the programmer to explicitly manage memory. This is in contrast to memory-managed languages that allocate and free memory automatically. This automatic management can be done through the use of a *garbage collector*, which keeps track of when objects are no longer needed and reclaim the associated memory automatically. Garbage collectors have been around since the late 50's in LISP. More recent languages that use automatic garbage collection are Java and Python. As mentioned earlier, Z is a garbage collector available for Java applications running on the HotSpot JVM, and the context for this work.

A managed language such as Java runs within a virtual execution environment, colloquially referred to as a "*runtime*". The runtime provides the programs with certain abstractions, and handles resource management tasks, such as allocation and deallocation. When the focus is on garbage collection, threads which execute

the user application are commonly referred to as *mutator threads*. From the perspective of the GC, we do not care what application threads do, just that whatever they do, it typically involves mutating the heap. Threads that are not executing program code but are executing garbage collection code are naturally named GC threads, short for garbage collection threads. This separation is needed to correctly discuss which code is being executed when and what assumptions must be made when the code is executed.

2.2 Pointers in Runtimes

Both GC threads and mutator threads commonly traverse the heap by following object pointers. *Roots* are pointers held in application storage which are directly accessible to mutator threads without going through other objects (e.g., stack variables). Roots point to *root objects*, and can intuitively be defined as being top-level in the graph of objects. All other objects are accessed by traversing paths of object fields starting from a root. The *lifetime* of an object is the time during program execution in which the object is reachable. Reachability is intuitively described as the existence of a path of pointers from some root through other objects, to an object. An object which is reachable is considered live [5]. Liveness and reachability are very important concepts for garbage collectors, as that is what determines what will be collected and not: all memory which is not reachable and hence garbage will be reclaimed.

Not all pointers are treated the same in the JVM. An *ordinary object pointer*, abbreviated "oop", is a pointer that points to a *managed object*, i.e. an object whose lifetime is managed by the runtime. Such objects can for example be subject to being moved or deleted by the garbage collector. A significant part of the memory footprint of a Java program running on the HotSpot JVM consists of oops, around 20% [4]. ZGC represents oops in a special way by putting metadata in their unused bits. The metadata is used for several things covered in Section 2.7.2. Amongst them is telling if an object is referred to by a *finalizer*, which is a function which may be called upon a heap-allocated object's destruction to allow graceful reclamation of resources prior to collection [3]. Knowing what objects have finalizers requires metadata, which restricts address range further, as discussed in Section 7. Finalizers have since long been deprecated, but must still be supported for back-

wards compatibility.

A garbage collector which follows the so-called *mark-compact* algorithm is considered *compacting*. Such a garbage collector relocates live objects to remove gaps in memory, providing better memory usage and performance [5]. Knowing what objects are slated for relocating also requires metadata. Furthermore, compacting garbage collectors typically divide the program heap into sections called *pages*, and compacts on a page-basis [5], see Figure 1. Z is a compacting garbage collector, which presents a problem in Section 6.5.2. A future release of Z is also *generational*. A garbage collector is considered generational if it discriminates between objects that have survived a collection cycle. This is based on the observation that most heap-allocated objects "die young", and those who survive live for a long time. Hence, promoting surviving objects to an "older" generation and collecting them less frequently can give a lot of performance benefits [5]. Typically, the young and old generations are collected separately. These are respectively called young and old cycles. This future version of ZGC is the version this project was built upon.

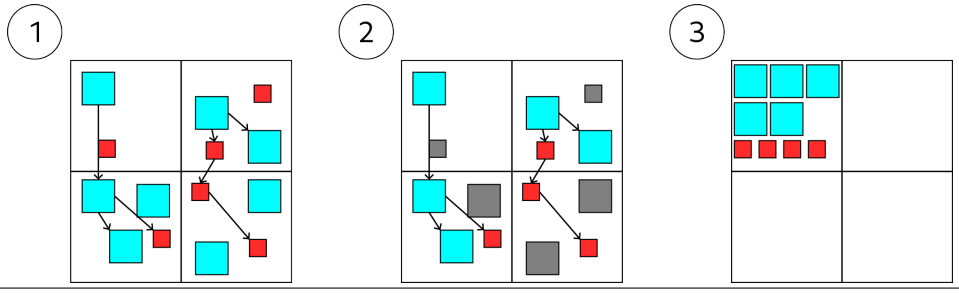

Figure 1 1: A heap divided into four pages has allocated objects in it.
2: The garbage collector has traversed oops and now knows what objects are alive.
3: The garbage collector has *compacted* all live objects and reclaimed the memory of dead objects.

Lastly, and unrelated to the above, an *abstract data type* (ADT) is a mathematical model for a data type, defining operations and states formally to reason about the data types semantics from a user's perspective. This is in contrast to a data type, which specifies a concrete implementation. Dale & Walker specify an abstract data type as a "class of objects whose logical behavior is defined by a set of values and a set of operations" [6]. The intuitive description is that an ADT is a bunch of functions on a type, defined in theory only. This is used in Section 6.5.1 to formally define semantics around a concurrent data structure. When regarding an ADT in a

concurrent setting, it's *executive point* is of interest: we can think of an operation as taking an amount of time, and at one point during the time interval the action of the operation has been "committed" or "executed".

2.3 Pointer Compression

Pointers in 64 bit architectures have an addressable range of 16 exabytes, up to 64 exabytes if individual byte addressing is disregarded, and 128 if only eight-byte alignment is needed. It is fair to say that the likelihood of a computer program using anything close to an exabyte is small, so most bits will always be zero. *Pointer compression* is the collective name of techniques for storing pointers using fewer bits to reduce memory footprint. Pointers are typically compressed to 32 bits due to word alignment [8].

Moving from 32-bit pointers to 64-bit pointers increased memory footprint by 40% for certain Java programs [8]. Looking at the V8 JavaScript engine, pointer compression saved around 40% memory, but at the cost of each web process being limited to 4 GB of memory [1]. It is clear that pointer compression can save a lot of memory.

A conventional way of implementing pointer compression is to represent the pointer as an offset from a known static point in the heap. This is opposed to representing it as an offset from 0, which could waste range as the heap may not be located there. See Figure 2.

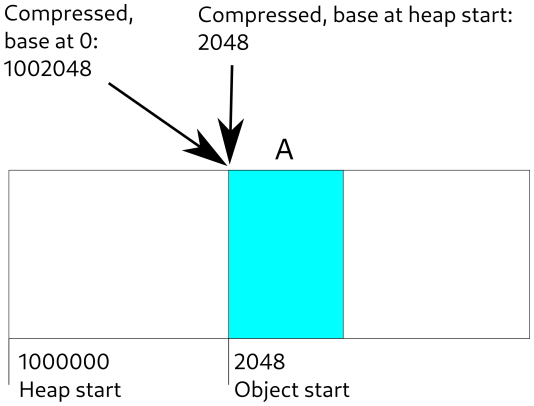

Figure 2 A compressed pointer can be stored as the literal address, i.e. an offset from 0, or from a fixed point in the application heap.

2.4 Object Relative Addressing (ORA)

An alternative way to represent a compressed pointer is to represent it as an offset from a dynamic point in the heap, typically the start of an object or field in which the pointer resides. This avoids the problem of address range as the range is now relative to each object, which would reduce the number of bits to express a pointer, assuming that the locality of objects is good, which is a reasonable assumption as compacting garbage collectors increase locality [5]. See Figure 3.

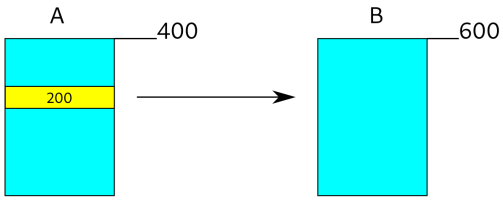

Figure 3 Object A has field containing an ORA pointer to object B. A resides at memory address 400, and B at 600. The contents of the field is then 200, the relative distance between A and B.

In the case that the distance between two objects is too big to fit in 32 bits, some type of fallback mechanism is needed. Previous work has registered pointer locations in a *Long Address Table*, a mapping between field addresses and the full 64 bit contents [8]. See Figure 4.

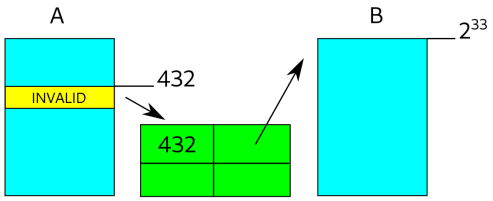

Figure 4 In this case, B resides too far away to encode as ORA. There is therefore a LAT entry which maps the field address to the full pointer which points at B.

2.5 The HotSpot JVM

A virtual machine is a process that emulates a computer setting for the sake of running other programs inside it. A typical example of virtual machines is running other operating systems as a process inside a host operating system. HotSpot is a virtual machine currently developed by Oracle, designed for running Java programs. HotSpot features *just-in-time* compilers, compiling often-executed code paths to native machine code, speeding up programs during runtime. HotSpot has three levels of compilation: the *interpreter* which reads Java bytecode as-is with very slight optimizations, the lightly optimizing *C1* compiler, and the heavily optimizing *C2* compiler. The C2 compiler performs certain optimizations like *inlining* lookups, i.e. performing the lookup and inserting the value it found as an immediate value instead of compiling the lookup itself. For example, it may look up and inline *field offsets*, covered below. This has the side effect of there being two versions of the same information: one which the interpreter reads by executing code which performs lookups, and one which is represented as immediate values in machine code. JITed code is stored in a separate piece of memory known as the *code cache*.

2.5.1 Object Layout in HotSpot

Every object in HotSpot belongs to a specific class, and each class has a *field layout* that describes how the fields of instances of the class are laid out in memory. Importantly, all instances of a certain class are described by the same *singular* layout, each object does not hold a copy of the layout. Furthermore, if class A inherits from class B, then A inherits the layout from B as well. Field layouts are defined in terms of their *offset* from a known point in the object's start in memory, i.e. a field F can start 32 bytes in, and the next field starts 40 bytes in, implying that F occupies 8 bytes.

Offset isn't the only metadata available regarding fields. Fields also have *access flags*. Access flags are field metadata used for signaling properties like if the field is static or volatile, as well as their type.

2.6 Pointer Compression in HotSpot

HotSpot uses an implementation of compressed pointers where a pointer is represented as an offset from the heap start as a base. The implementation circumvents the problem of 32 bits only being able to represent a 4 GB range by exploiting that objects in the 64-bit JVM are 8-byte aligned. If we think of the offset as possible object locations rather than byte addresses, the address range is multiplied by 8, resulting in 32 GB addressable memory. HotSpot's implementation of compressed oops is dubbed *NarrowOops*, and is not supported by ZGC due to the HotSpot implementation clashing with the intended use case of the garbage collector, described in detail below.

2.7 The Z Garbage Collector

The Z garbage collector is a concurrent garbage collector found in HotSpot, released as production ready in September 2020 as part of the JDK 15 release. Quoting the OpenJDK wiki, ZGC is concurrent, region-based, uses coloured pointers as well as load and store barriers, and has goals of sub-millisecond pause times with heap sizes ranging from 8 MB to 16 TB [7]. These features and goals all prove obstacles when implementing compressed pointers. What follows in this subsection is a description of these terms, followed up by how they clash with compressed pointers.

The ZGC release in OpenJDK is a single-generation GC. Due to large upcoming changes in ZGC, namely the addition of *store barriers*, this project has used a development branch of ZGC which adds support for multiple generations. Following descriptions hence may not apply to ZGC as released in JDK 18, which as of this writing is the most recent release. More on this in Section 5.

2.7.1 Concurrent Collection

Concurrent collection is garbage collection work performed without stopping the user application. The main challenge of concurrent collectors is to coordinate mutator work and GC work simultaneously on the same object. For example, if a GC thread moves an object, it is imperative that a mutator does not follow a pointer

to the object that has not yet been updated, as the results of such an operation is undefined. One key ingredient in the coordination of such concurrent activities is by adding so-called *load and store barriers* into compiled code, a stopping point for mutator threads, in which it is forced to synchronize its operation with the GC whenever reading or writing to oops on the heap. This is elaborated on below.

A garbage collection cycle in ZGC consists of three phases. These three phases are the *mark-and-remap phase*, the *prepare-for-relocation phase*, and the *relocate phase*. The latter two phases begin with a sub-millisecond *pause* where all mutator threads are forced to stop executing, although this is subject to change. An important aspect of the pauses is that their time complexity is $\mathcal{O}$(n) in relation to the number of mutators, but $\mathcal{O}(1)$ in relation to the amount of live objects on the heap as well as heap size. This will present a problem described later in Section 6.5.2, where pauses can't be used to avoid concurrency issues.

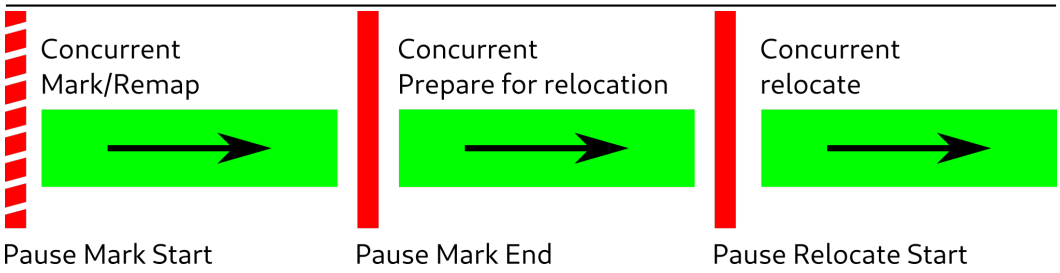

Figure 5 Visual overview of ZGC phases as of JDK 18. Note that the start-of-mark phase pause will be removed in a future release and was therefore considered as unusable for the sake of this project.

The mark-and-remap phase traverses and keeps track of objects by traversing through the graph of live objects, starting with the roots. As the name suggests, the phase also *remaps* oops. Remapping means to correct oops whose objects have been relocated but the pointer hasn't been updated.

The prepare-for-relocation phase does several things, but only *relocation set selection* is interesting for this project. The mark phase gathers information on how full pages are, which is utilized to determine which pages are suited for compacting. Pages which are considered candidates for compacting are added to the *relocation set*, and as objects are relocated from these pages, their old and new addresses are registered in a *forwarding table* that is used to remap pointers to moved objects as part of dereferencing them. The relocation set is a set of pointers of pages full

of objects to relocate. The forwarding table is a table of the from-addresses to to-addresses, but at this stage the to-addresses aren't necessarily filled, as objects may not have been relocated.

The relocation phase utilizes the populated relocation set and forwarding table to relocate objects. Most of the relocating is done by GC threads. However, if a mutator touches an object on a page slated for relocation before the relocation has taken place, the mutator must itself perform the relocation (of the object in question) before commencing its access. Notably, the relocation never remaps pointers, but rather registers them in the forwarding table for remapping during the next mark phase or barrier. After the relocation phase pause, what pointers are considered "good" (covered below) has changed such that all pointers are now considered "bad".

2.7.2 Coloured Oops

A coloured pointer is a pointer where parts of the bits are used to store metadata, nicknamed *colours*. In generational ZGC, all oops are coloured, and this metadata is divided into four sections occupying 12 bits. The colours are as follows:

- Four *remembered* bits. An object may be part of the *remembered set* if it is part of the old generation and may contain pointers to the young generation. The remember set is nicknamed "remset". Remembered pointers allow the garbage collector to correctly determine liveness in the young generation without scanning the old generation. The remembered bits are set after a young collection cycle to indicate that the pointer needs adding to the remset, and cleared when it is added.
- Two *finalizable* bits. These bits are used to indicate if an object is reachable by a finalizer. If an object is only reachable by a finalizer, it can't be collected yet[1].
- Two *marked young* bits. These bits indicate if a young cycle has marked this object.

[1] Providing a reference for "soon-to-be" dead memory to an arbitrary piece of code is in hindsight considered a bad move.

- Two *marked old* bits. These bits indicate if an old cycle has marked this object. The reason for having two sets of bits, one for young and one for old, is to allow both a young and an old cycle to occur simultaneously. If an old cycle finds a young object, it is added as a root to the young cycle.
- Four *remapped* bits. These bits are used by the barriers to recognize if the object has been moved and the pointer needs remapping. Remapping is done by finding the appropriate forwarding entry in the forwarding table. If there is no to-entry in the forwarding table but the remap bits indicate that the oop needs remapping, it indicates that the object needs relocating, and the mutator thread will claim and perform the relocation.

Finally, there are four reserved bits at the end, for a total of 16 reserved bits. See Figure 6 for a visual overview.

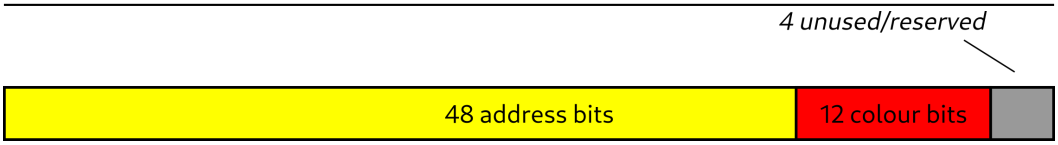

Figure 6 Visual description of coloured pointers in the in-development version of ZGC.

2.7.3 Load and Store Barriers

As mentioned in Section 2.7.1, ZGC solves parts of it concurrency issues through so-called *load and store barriers*, intuitively described as stopping points for mutator threads. The barriers are there to ensure that dereferencing a pointer is memory safe and won't cause memory corruption. This is determined by reading the colors. Colours can be *good* or *bad*. If colours are bad, the mutator thread is forced to go through a *slow path*, healing the pointer to make it good. This slow path, i.e. healing, can for example be putting the pointer into the remembered set, or remapping it according to the forwarding table. The corresponding *fast path* simply returns without any healing. For stores, colours are considered bad when remapping is needed, the pointer needs marking, or the pointer needs adding to the remset. Colours are considered bad for loads if the object needs remapping and relocation. Colours are considered good when they are, surprisingly, not bad.

2.7.4 Why ZGC? Why pointer compression?

The intended use case of ZGC is latency-critical applications using large heaps that are fine with the possible tradeoff of lower throughput. Think web services: sending a request only to wait 1000 ms extra because the server process just triggered a huge garbage collection cycle is worse than suffering another 10 ms on every request for end users. Depending on the number of users and what type of load the web service is intended to support, it may want very large heaps as well.

Pointer compression saves space and leads to smaller objects. Smaller objects make better usage of RAM or lower the memory requirements of the program, and increases locality and hence performance. In short, pointer compression is desirable because saving memory is good.

3 Challenges with Pointer Compression for ZGC

With the above in mind, problems arise for implementing pointer compression for ZGC. Observe that when writing memory on x86 architectures, the memory is usually aligned in 32-bit chunks to align with the computer words. Hence, we assume that we will compress pointers to 32 bits, although this is discussed in Section 7. With address range being limited to 32 GB when using 32 bits, the goal of multi-terabyte heaps is not met, resulting in half the selling point of ZGC, namely big heaps, being thrown out the window. The need for metadata imposes a lower bound on pointer compression. Allowing all 12 colours restricts address range to 7.8 MB, which is under the 8 MB ZGC requires. This problem could be mitigated if object-relative addressing is used, exploiting that objects that reference each other are often close in memory, as is the goal of compacting garbage collectors. The problem with object relative addressing is that it depends on runtime information which isn't always readily available and is subject to change, such as the start of the next object.

It is here we arrive at using object relative addressing, but compressing *after* objects have already been allocated, therefore requiring on-the-fly conversion. It is not feasible to convert during pauses, as the pause times would increase with live object size, which is undesirable for ZGC. The other alternative is to convert during a

concurrent phase, such as the mark phase. In that case, there is a problem where two objects of the same class could follow differing layouts.

4 Related Work

4.1 Pointer Compression in V8

The development team for the V8 JavaScript engine (notably found in the Chromium and Chrome web browsers) implemented pointer compression, detailed in their 2020 blog post [1]. The implementation consists of base compression from a known offset, which is acceptable as most V8 embedders use less than 32GB. For example, Chrome uses 4GB. 64-bit pointers in the V8 engine use the two least significant bits for value tagging, as opposed to ZGC's 12 bits. The compression scheme used considers base compression where the base is at the start of the heap. The authors claim that most of the runtime overhead can be removed on ARM and x86 architectures through compiler optimizations, notably by reducing the amount of emitted instructions and disregarding unimportant memory. The V8 compiler is somewhat similiar to the C2 compiler in the HotSpot JVM, as it also uses a *sea of nodes* strategy for finding compiler optimizations, but most of the optimization they did relied on the specific layout and details of the V8 engine and how it represents so-called *SMIs*, signed integers.

4.2 ORA for 64 bit Java

Venstermans et. al. showed 2007 that their ORA implementation doesn't induce any statistically significant slowdown on Java applications running on the the Jikes RVM (which is a different JVM from HotSpot) but reduces memory consumption by around 10%. The implementation considers compressing 64-bit pointers to 32-bits by storing it as a relative offset between holder and object, and storing an entry in a *Long Address Table* in case the offset is too big. The authors solve the issue of recognizing if a pointer is compressed or not by using a metadata bit in the least significant bits, limiting potential address range. To prevent the LAT-table from growing infinitely the authors suggest managing the LAT-table during mark-sweep

cycles, which is trivial as the garbage collector used stops mutator threads during garbage collections. The authors suggest that pointers are compressed during runtime, but the paper never details how to compress the resulting voids left in objects after compression is done.

5 Method

This section describes the method of working. The goal of the project is as stated not necessarily to produce a complete implementation or algorithm, but to explore the design space and realize what problems may stand in the way.

5.1 Exploratory Programming

The research performed here was done using a small sample Java program and modifying the JVM codebase, recompiling and debugging using GDB to find what problems occur when certain details are implemented. The test program instantiates an object of a custom class, reads and writes to the objects fields, then calls garbage collection, and then does reads and writes references again. This acts as a large unit test: read and write an oop, ensure that it is compressed, and try to read and write to the field again. The expected behaviour is that nothing crashes on reads, writes or compression, and that the data stored is available. A tool that was used is Mozilla's `rr`, a program which records execution and allows deterministic debugging as the recording will be the same. It also allows easy reverse execution.

The branch used for development was a future unreleased version of the generational branch of ZGC with helpful patches by Jonas Norlinder, which is part of his work on speculatively compressed pointers.

5.2 Delimitations

The following delimitations were placed to reduce the scope of the project:

- The prototype produced here disregards the use of the C1 and C2 compilers,

and rather only considers the interpreter. If C2 was enabled and the code for accessing a class was emitted to machine code, it would have needed to be changed or retired when the compression happens after emitting. This delimitations hence avoids the need to consider retiring emitted machine code.

- The prototype only considers user-implemented classes which no other class inherits from. Furthermore, it won't compress the entire inheritance chain, but rather only the ”bottommost” level. A class which inherits from some other class also inherits it's layout. Changing a parent class could result in the inheriting class having a differing layout for the same data. This delimitation hence avoids the problem where other classes would inherit a field layout that could later on be rendered invalid.
- The task of finding a heuristic to determine what classes are to be compressed has been steered clear of, as the question was deemed large enough to warrant an entire thesis project. Instead, it is assumed that there is a pre-filled datastructure containing classes which are candidates for compression.
- There is an issue in encoding and de-encoding the null pointer. In HotSpot running ZGC, the null pointer is represented as a literal zero. However, an object is also allowed to be placed at zero offset from heap base, i.e. start of the heap. Hence, if one were to remove the heap base from a pointer for the sake of compression, it would be impossible to distinguish the null pointer from a compressed oop pointing at the start of the heap. Therefore the base heap address is prohibited from containing an object.
- Arrays are a special case which is not considered for compression due to lack of time.
- To keep things simple, the implementation only consider the x86 architecture, and makes assumptions that computer words are 4 bytes and that pointers occupy 8 bytes.

6 Results

This section contains the result, which is a prototype algorithm and implementation. What follows is an algorithm description and a discussion of discarded

approaches.

Consider a program with a user-defined class A that has no subclasses, and at least one non-static non-primitive member. Then, A has at least one field containing an oop in its memory layout. Assume that the JVM has deemed the oop fields of A to be suited for compression. First, an overview of what happens next.

6.1 Rearranging and Reconstructing Field Layout

During the next relocation phase pause, the layout of A will be altered. Signifying that each field now occupies four bytes instead of eight is a matter of changing their type to one that is 4 bytes wide. Changing the type of the field turned out to be a bad idea. Instead the fields access flags are changed to signal that it is 4 bytes wide. See Algorithm 1

for C *in compressable classes* **do**
  for f *in* $oops(C)$ **do**
    $A_f \leftarrow A_f \mid$ *compressed bit*
  end
end

Algorithm 1: The algorithm after the first step. The flags A_f for each oop-field f gets a bit set which signals that it is four bytes wide.

Out of the three pauses in the ZGC cycle, layout altering happens in the relocation phase pause for four reasons. First, in a future release the prepare-for-relocate pause will be removed. Second, consider trying to change the layout of a class in the concurrent part of a phase. A race condition occurs where a thread may be trying to access the an object according to an updated layout to which it doesn't adhere to. Changing the layout in the relocation pause leaves no room for inconsistency in layouts as mutator threads are stopped, as opposed to changing layout during the concurrent parts of a phase. Third, as objects aren't actually compressed until relocated it is in theory better to change the layout as close to relocation as possible, so that GC threads claim more of the work of relocating. This is because GC threads typically don't perform relocation until after a relocation phase, leaving mutator threads with relocation work, and hence not executing application code. Altering layout also leaves newly allocated objects already adhering to the layout, as opposed to an object being allocated while the layout is changing. Last,

after the relocation pause, what colours are considered bad changes, so all pointers has to go through a slow path, ensuring that the opportunity to compress objects are presented.

With the fields now being treated as 4 bytes wide, the layout needs reconstructing, as there would now be a gap between the end of fields and what comes after. Reconstructing the layout is a matter of iterating over layout elements and placing them tightly, using their *type sizes* to determine where the next element should be placed. This is also done during the pause. See Algorithm 2.

```
for C in compressable classes do
    for f in oops(C) do
        A_f ← A_f | compressed bit
    end
    S ← 0
    for f in fields(A) do
        O_f ← S
        if A_f & compressed bit then
            S ← S + 4
        else
            S ← S + size(F)
        end
    end
end
```

Algorithm 2: The algorithm after the second step. Observe that the first nested loop is from Algorithm 1. The offset O_f for each field f gets set to the previous total offset S, which is then incremented by the type size $size(F)$ of f. Observe that this is done over *all* fields, as opposed to the access flag mutating which is only done over oops.

6.2 Compressing Object Instances

So far, we have changed how the class describes the layout of the objects on the heap. But we have not changed the actual objects so that they confirm to the layout description. Each and every object of the class must be added for relocation at the same time. This is currently done by adding all regions for relocation at the same time, more on this in Section 6.5. Hereon, each individual object relocation will compress and realign the affected oops to their 32-bit object-relative representation, making sure that the objects comply with the layout post-relocate. Keeping in

mind that the layout has been changed without preserving the previous layout, the contents have to be compressed through pointer arithmetic. See Algorithm 3.

for f_i *in* $oops(C)$ **do**
  $f_{old} \leftarrow f_0 + 8 \cdot i$
  $f_i \leftarrow (\text{offset}(f_i) - \text{offset}(f_{old})) \ll 12 \,\&\, \text{colours}(f_{old})$
end

Algorithm 3: The algorithm for actually compressing the pointer fields. f_{old} denotes the field address where the oop actually resides in the object being inspected, as it does not actually align with its layout. The function offset(f) denotes the pointer or oop offset of the contents of a field f from heap base. The function colours(f) denotes the colours of an oop residing in a field f.

It is important that no object that has had its layout changed is accessed as if it was adhering to its layout prior to being compressed, as the reads would then be undefined. This is only possible if all instances of a compressed class are set to go through a slow path, (recall Section 2.7.3) where they are compressed. The current design suggests doing so during a relocation, which is relatively cheap as the object is being copied anyway. An additional benefit to utilizing relocation is that relocation is a critical section with synchronization, i.e. two threads can't race to relocate to the same memory, so concurrency issues are eliminated. Making sure that all objects of a class is subject to relocation does however present its own problems: finding and adding all pages which contain such an object to the relocation set during the pause would make it scale with the live object set. The current design adds *all* pages to the relocation set, for simplicity. This is further discussed in Section 6.5.

6.3 Encoding and Decoding

The chosen implementation relies on inspecting the field address or underlying value to determine if a field is compressed. Encoding from 64 to 32 bits includes storing the colours at the least significant bits without the reserved four lower bits, and storing the relative offset from the field to the object as an eight-byte aligned distance using twos complement as representation.

A field is deemed compressed if one of the following conditions are met:

- If the field address is four-byte aligned, as full-width fields would be eight

byte aligned.

- If any of the lower bits 1, 2, or 3 are set on the oop. As the colours are stored without their reserved zero bits, at least one of them will always be set on compressed pointers. Bit 0 can't be checked as its used to mark oops during initialization, and would give false positives.
- If the lower 32 bits of a field is a *raw null*, i.e. offset 0 with no colors *and* there is a fallback registered for this field. Registering a fallback for a field implies adding the uncompressed value of the field to the textitlong address table (recall Section 2.4), with the field address as key. More on this below.

Registering fallback on null is important. Consider two fields laid out next to each other in memory, both containing a raw null. Trying to figure out if the field is 32 bits or 64 bits wide based on it's content is impossible, as it's all zero. Registering a fallback for raw nulls when compressing solves this issue, as only 32-bit wide fields have fallbacks.

With 32 bits, 12 being used for colors, and storing offsets as eight-byte aligned, the relative address range is $2^{23} - 1$, or about 8 MB. Assuming that a compressed pointer requires 4 bytes, a non-compressed pointer requiring 8 bytes, and a compressed pointer with a fallback requires 12 bytes, at most half of the amount of oops that are eligible for compression can have a fallback before this compression scheme doesn't save space. See Appendix A for calculations. As there is also a runtime overhead on compressing, the space savings need to be sufficiently large to be worth it. Profiling ZGC reveals that for certain processes, about 70 % of offsets are larger than 8 MB when using a 4 GB heap. This suggests that 8 MB relative range is far from sufficient. See Appendix B for detailed results.

6.3.1 Fall-backs & Concurrency

As the oop field and the long address table are two different memory locations, it is prone to concurrency issues. Consider the following: mutator thread A is trying to access a compressed field which is being written to by mutator thread B. The field has a fallback, but it will be removed as the offset being written by B is sufficiently small. B has two possible ways to do this. Write the field contents and then clear

the fallback, or clear the fallback and then write the field contents. In either case, A is presented the opportunity to do the wrong thing: either note that there is no fallback and read incorrect field contents as B has cleared but not yet written, or see that there is a fallback which is incorrect as B has written but not yet cleared. A solution to this is suggested in Section 6.5.

6.4 Discarded Approaches

This thesis is as stated about exploring the design space for dynamically compressed pointers. Naturally, the proposed solution is not the first one thought of. This section details two other approaches that were discarded.

6.4.1 Using Access Flags to Denote Compression

Each field has a set of *access flags* which encode important information used when accessing fields, such as if the field is volatile or static. One approach attempted to add an access flag that signals that the field is compressed, and make the interpreter respect the access flag by emitting a code path that branches to separate calls for 32-bit wide oops depending on the flags. This approach proved problematic on three points. First, access flags are not easily accessible from the interpreter, instead they are accessed through a cache known as the *Constant Pool Entry Cache*. This cache is a compact intermediary representation for access flags. The code is hardly documented, and programming towards the cache entry interface proved a challenge in and of itself; there is seemingly several constant pools and caches for a class, revealed by inspecting the memory address of the cache being mutated. Inspecting the memory address found in the interpreter can be done using `gdb` and cycling to the register layout, using `layout next`.

Second, as it is machine code being emitted there is no way to use overloaded calls, resulting in two nearly identical but slightly different manually written code paths. This complicates the interpreter code a lot.

Last, having this check at the interpreter level is too ”high” - there is lower level garbage collection code that is then unaware of the notion of compressed fields. The access field information is also discarded on the way down to this code, so

the lower level garbage collection code would need it's own implementation. This proved to be a significant part of the ZGC codebase, including the implementation for remsets and relocation sets. Dealing with compression at a lower level solves this problem.

Changing the access flag is still done, as mentioned in Section 2.5.1, but only to signal that the field is 4 bytes wide.

6.4.2 Mark Phase & Load Barrier Compression

A previous implementation explored performing compression during the concurrent mark phase as the mark phase visits every object when traversing the object graph to determine the liveness of objects. Upon visiting an object of a class A that is to be compressed, *set* the objects class to a new class A', which is A but with a layout of compressed oops, and compress the object when a mutator thread takes the next slow path. This proved problematic for two reasons. First, creating new classes dynamically is hard: there are no APIs for creating a class which mimics another, and internal datastructures such as *class loaders* and *constant pools* have fixed sizes which would need manual reallocation to fit new classes. In short, the JVM isn't made for creating classes dynamically. Without being able to change the class of an object the aforementioned problem of objects having inconsistent layouts would occur. Second, there's a race condition in compressing, as two threads can both enter slow paths. This would require synchronization - more on this in Section 6.5.2. In this approach, determining if fields are compressed would use a similar approach to the suggested one.

6.5 Future Work

The following are problems which were dodged and no solution has been attempted. As such, all suggestions are speculative. They need to be solved before the implementation can be profiled and tested.

6.5.1 Solving Concurrency & Fall-backs

As mentioned in Section 6.3.1, there is a concurrency issue regarding registering that a field has an entry in the long address table. The issue is multifaceted - there is no abstract datatype ”compressed oop-field with potential fallback”, so the semantics regarding concurrency is poorly defined. For example, is reading a stale fallback pointer invalid or acceptable? For this thesis, we can define our own abstract datatype (F, b), which is a compressed oop field F with potential fallback field b. It has the following operations:

- $store(F, b, O)$ compresses the address O and stores it in F. If O can be compressed and b is not false, an invalid pointer is stored in F, b is set to false, and compressed O is stored in F. If O is compressible and b is false, compressed O is stored in F. If O is not compressible, it is stored in b, and an invalid pointer is stored in F. The executive point of *store* depends on what it does: if O is compressible and b is not false, the executive point is when b is set to false. If O is compressible and b is false, the executive point is when compressed O is stored in F. If O is not compressible, the executive point is when an invalid pointer is stored in F.
- $read(F, b)$ reads b and returns it's contents, decompressing if it is compressed. If b is false, it decompresses and returns the contents of F. If the content of F is invalid, *read* tries again. The executive point of *read* is either when it reads b and it is not false, or when it reads F.

The important part of this definition is defining that reading a pointer whilst a fallback exists is *not* invalid as long as the contents of F is valid. Hence the problem described in Section 6.3 can be mitigated: B would either read the old pointer, which is valid, or read an invalid pointer and then try again until either the fallback of F exists or the pointer in F is no longer invalid. Implementing this abstract data structure does however rely on a concurrent implementation of the fallback mechanism, possibly using a concurrent map, as it relies on an atomic 'get-or-false' operation on the fallback field.

6.5.2 Properly Ensure Relocation

The current design dodges the problem of ensuring objects are compressed by adding all pages to the relocation set, thereby relocating *everything*. There is a problem in that we can't scan for objects during the pause as such a scan operates linearly with the number of live objects. An alternative is to utilize that all pointers are bad after relocation and offload compression to slow-paths. This requires synchronization such as locks or mutexes, as two threads can enter slow-paths at once. To reduce contention on locks, we can utilize an array of locks that scale with the number of threads, and map field addresses to locks. This spreads out contention to many locks. Using for example queue locks would further reduce contention.

Performing compression in slow paths would solve an additional problem. If Z during relocation finds a page which is compact it is possible that it won't relocate its objects, but rather consider the page part of the old generation, resulting in objects not being compressed as they are not relocated. This is avoided if compression is performed during slow paths.

6.5.3 Using the Encoding and De-encoding API Everywhere

Certain parts of the JVM code currently utilizes an API that doesn't respect speculatively compressed oops, and would mangle memory if it were to write to fields which were compressed. The main example is *string tables*, a method for *string deduplication*, which stores oops through the so-called *native* back-end, unaware that it may be handling fields that have a different bit representation. This back-end need to be made aware of speculatively compressed oops.

6.5.4 C2 implementation

The C2 compiler *inlines* offsets to fields when accessing them. This means that instead of lookup the offset up on each read, which the interpreter would do, they are instead looked up once and emitted as immediate values in the machine code. Hence, changing the layout of a class when running C2 would require modifying the emitted code, or perhaps discarding the emitted code and re-emitting it entirely.

6.5.5 Compressing Inheritance Chains

The design currently ignores inherited fields, and only compresses classes that do not inherit from other user-defined classes. Trying to compress fields of a class that both inherit and is a parent to some other class would leave gaps in the layout with the current design. Then, if a super class has shrunk, the layout of a subclass would be shifted back by the equivalent amount of bytes. Another potential solution could be found in the heuristic. Ensuring that entire class chains are deemed to be compressed at the same time could alleviate this problem.

7 Discussion

The proposed design has a lot of outstanding issues and can as of currently not measure how severe the runtime impact will be and how much memory usage will be reduced. This project proved very hard. The majority of the complexity is found in that only certain classes are to be compressed with tough requirements on performance, *after* many objects of said class has been allocated and are alive. We can imagine an implementation where a decision is taken very early in program execution which states that certain classes are now compressed and use ORA prior to any of it's objects being allocated. This way the complexity of managing layouts and objects that may not conform to said layout is removed. In that case an opportunity for optimization would be lost. Knowing object layout in memory, whatever heuristic that decides what classes are suitable for compression could avoid classes with the potential for fallbacks.

An approach that could eliminate the need to use ORA and all it entails is to use something other than 32 bits for compression, perhaps 48 bits, at the cost of doing unaligned accesses, which are typically expensive. If the range can be made sufficient without ORA, the decision to use compressed oops can be made statically. The drawback is that the maximum grade of compression is reduced. Furthermore, to achieve a range of 4 TB a pointer would need 42 bits of address. With 12 bits for colours and utilizing the same 8-byte alignment trick that normal compression in HotSpot uses, pointers would be compressed to 51 bits, which may not be worth it. Being able to compress colours as well could improve compression grade, seeing

as both the mark young and mark old colours are two bits whilst perhaps only one is needed could improve the compression grade.

Regarding address range, 8 MB seems insufficient. The first foray to get more address range is to see if the number of colors can reduced, or if the colors can be compressed. For example, the mark bits could perhaps be two bits instead of four, and the remapped bits could possibly be fewer. Beyond that, it is possible to force objects to be aligned to for example 16 or 32 bytes, resulting in more bits always being 0, so address range can be increased. Ideally, over 1 GB in address range would be needed, which is equivalent to 30 bits of address range. Currently, there are 23 bits of address range to be used, assuming 8-byte object alignment.

8 Conclusion

This project explored the design space for implementing compressed oops for ZGC, and finding where the problems occur. The result is a design in which compressed pointers have a differing bit layout compared to uncompressed pointers, removing the four unneeded colours, and storing the address as an object-relative offset. The design uses the relocation phase pause found in ZGC to change the layout of affected classes and forces them to be relocated, at which point the actual objects are compressed. There are a number of outstanding problems: fallbacks for the ORA representation doesn't have a working concurrent implementation, ensuring that all relevant objects are actually compressed needs a more efficient implementation, fallbacks aren't defined for a concurrent mode of operation, certain parts of the JVM code is unaware of the notion of dynamically compressed pointers as implemented here, and address range is insufficient.

A Fallback Ratio Calculations

This calculation is used to determine how many compressed oops can have a fallback and still save space. This is used in 6.3

Assume x is the amount of oops in a running application. Assume that one uncompressed oop occupies 8 bytes, a compressed oop 4 bytes, and a compressed

oop with fallback occupies 12 bytes. Assume also that all pointers are either compressed or compressed with a fallback. How many compressed oops, expressed in a percentage of all oops a, can have a fallback before it is no longer worth it to compress any pointers, storage-wise?

The equation is set up as follows. Assume all oops will be compressed. The space occupied by uncompressed pointers is $8x$, as they occupy 8 bytes. The space occupied by compressed pointers with fallback is $12ax$, as they occupy 12 bytes, and is a percentage of the total amount of oops. The space occupied by compressed pointers without fallback is then $4(1-a)x$, i.e. the rest of the oops, which occupy four bytes. Solving for a gives:

$$
\begin{aligned}
8x &= 4(1-a)x + 12ax \\
8 &= 4(1-a) + 12a \\
8 &= 4 - 4a + 12a \\
4 &= 8a \\
\frac{4}{8} &= a \\
a &= \frac{1}{2}
\end{aligned}
$$

Hence, at most half of the compressed oops can have a fallback before space is no longer saved.

B Benchmarking Pointer Distances

Fallbacks are created when writing a pointer with too large an offset to be represented in the amount of address bits available. A benchmark was performed to inspect what amount of pointers would contain a fallback, had it been compressed. A modified build of the OpenJDK JVM was used, where on each oop write it would print the field address and if the field would have a fallback. This was written to a file. The benchmark ran was the DaCapo 9.12 Jython benchmark using a 4 GB heap [2]. The benchmark was run on a Thinkpad T14 with 16 GB of RAM and a

Ryzen Pro 4750U CPU, running Fedora 36. A python script, attached below, filters unique addresses and reports how many fields contain offsets over 8 MB as a percentage of the total. The result shows that over 70 % of pointer offsets are over 8 MB and would hence need a fallback, which wouldn't save space. Below is the script and an example start.

This test inspects *every* oop-write and checks it's range. As stated in the report, a heuristic could choose what classes to compress such that the amount of fallbacks is minimized. Furthermore, this test was performed on a debug build, as the code wasn't working on fully optimized builds. Reproducing these results is difficult as they rely on a future, internal build of ZGC.

Listing 1: Example of running the benchmark and gathering results.

```
$ ls | grep output.txt   # No output file yet
$ java -version          # In-development build
openjdk version "18-internal" 2022-03-15
OpenJDK Runtime Environment (slowdebug build
18-internal+0-adhoc.linusshoravi.zgc-open-scoop-new)
OpenJDK 64-Bit Server VM (slowdebug build
18-internal+0-adhoc.linusshoravi.zgc-open-scoop-new, mixed mode)
$ java -XX:+UseZGC -Xmx4g -Xms4g -jar dacapo-9.12-MR1-bach.jar jython
# ... DaCapo output ...
$ ls | grep output.txt
output.txt               # Output file
$ head output.txt        # Each row is an address and it's offset.
0x40000000688,1672
0x40000000670,1648
0x40000000738,1672
0x40000000720,1824
0x400000007e8,1672
0x400000007d0,2000
0x40000000898,1672
0x40000000880,2176
0x40000000948,1672
0x40000000930,2352
$ python countoffets.py  # Finds % of unique field addresses
                         # with offsets over 8 MB
0.7179688716245731       # About 72% is over 8 MB
```

Listing 2: Script which counts offsets.

```
import pandas as pd
```

```
import numpy as np
import scipy.stats as stats

df = pd.read_csv("output.txt", names=['address', 'offset'],
                 low_memory=False)
# Note that we keep _first_ unique occurence here.
# Changing to 'last' actually makes the result worse.
df = df.drop_duplicates(subset='address', keep='first')

eight_mb = 8388608
over_eight_mb = df[df['offset'] > eight_mb]

print(over_eight_mb.shape[0] / df.shape[0])
```

References

[1] (2020) Pointer compression in v8. [Online]. Available: https://v8.dev/blog/pointer-compression

[2] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann, "The dacapo benchmarks: Java benchmarking development and analysis," in *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, ser. OOPSLA '06. New York, NY, USA: Association for Computing Machinery, 2006, p. 169–190. [Online]. Available: https://doi.org/10.1145/1167473.1167488

[3] H.-J. BOEHM, "Destructors, finalizers, and synchronization," vol. 38. Broadway, NY: ACM, 2003, pp. 262–272.

[4] R. Bruno, V. Jovanovic, C. Wimmer, and G. Alonso, "Compiler-assisted object inlining with value fields," in *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, ser. PLDI 2021. New York, NY, USA:

Association for Computing Machinery, 2021, p. 128–141. [Online]. Available: https://doi.org/10.1145/3453483.3454034

[5] R. Jones, A. Hosking, and E. Moss, *The garbage collection handbook: the art of automatic memory management*. Boca Raton, FL: CRC Press, 2012;2011;2016;.

[6] H. M. W. Nell Dale, *Abstract Data Types: Specifications, Implementations, and Applications*. Jones Bartlett Learning, 1996.

[7] Oracle. (2022) Openjdk wiki. [Online]. Available: https://wiki.openjdk.java.net/display/zgc/Main

[8] K. Venstermans, L. Eeckhout, and K. De Bosschere, *Object-Relative Addressing: Compressed Pointers in 64-Bit Java Virtual Machines*, ser. ECOOP 2007 – Object-Oriented Programming. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, vol. 4609, pp. 79–100.