FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Generational and Parallel Garbage Collection

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Dominik Inführ, BSc
Matrikelnummer 0925697

an der Fakultät für Informatik
der Technischen Universität Wien
Betreuung: Ao.Univ.Prof. Dipl.-Ing. Dr. Andreas Krall

Wien, 1. Juni 2019

Dominik Inführ	Andreas Krall

FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Generational and Parallel Garbage Collection

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering & Internet Computing

by

Dominik Inführ, BSc
Registration Number 0925697

to the Faculty of Informatics

at the TU Wien

Advisor: Ao.Univ.Prof. Dipl.-Ing. Dr. Andreas Krall

Vienna, 1st June, 2019

Dominik Inführ	Andreas Krall

Erklärung zur Verfassung der Arbeit

Dominik Inführ, BSc
Ringstrasse 25/2, 3062 Kirchstetten

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 1. Juni 2019

Dominik Inführ

Danksagung

Zunächst möchte ich Professor Krall dafür danken, mich durch diese Arbeit hindurch begleitet zu haben. Zahlreiche Treffen haben diese Arbeit nicht nur ermöglicht, sondern auch erheblich verbessert. Zusätzlich möchte mich auch bei ihm bedanken, mich durch seine Vorlesungen für das Feld der Programmiersprachenimplementierung begeistert zu haben.

Zuletzt möchte ich meinen Eltern und meiner Partnerin danken. Dafür dass sie mich bereits so lange unterstützen und mich immerdurch motiviert haben, diese Arbeit zu vollenden.

Acknowledgements

First I want to thank my advisor Prof. Krall for mentoring me and providing invaluable feedback throughout this process. Our many meetings discussing my work has certainly improved this thesis substantially. In addition I also want to thank him for introducing me to the field of programming languages, compiler construction and virtual machines through his various lectures at the university.

Last but not least, I want to thank both my parents and my partner. I am grateful to all of them for their continuous support and also for motivating me to finish this thesis.

Kurzfassung

Viele Programmiersprachen verwenden Tracing Garbage Collection (GC) zur automatischen Speicherverwaltung. Der GC gibt nicht mehr erforderlichen Speicher automatisch wieder frei, allerdings hat die Applikation selbst kaum Einfluss darüber, wann dies erfolgt. Der GC pausiert die Anwendung spätestens, wenn kein freier Speicher mehr verfügbar ist. Dazu berechnet der GC den transitiven Abschluss aller von der Anwendung erreichbaren Objekte im Arbeitsspeicher. Speicherbereiche, die nicht mehr erreicht werden können, werden vom GC wieder freigegeben. Im Anschluss daran kann die Ausführung der Applikation fortgeführt werden.

Dora ist eine Laufzeitmaschine für eine statisch getypte Programmiersprache. Funktionen werden bei ihrem ersten Aufruf in Maschinensprache übersetzt, ohne vorher interpretiert zu werden.

Diese Arbeit präsentiert den auf Generationen basierenden Speicherbereiniger *Swiper*. Dazu wird der Speicher in zwei Teilbereiche aufgeteilt: neue und alte Generation. Dies basiert auf der Annahme, dass die meisten Objekte relativ früh sterben. Objekte werden zunächst in der neuen Generation allokiert und werden in die alte Generation verschoben, sobald sie genug Speicherbereinigungen überlebt haben. *Swiper* implementiert zwei verschiedene Arten von Speicherbereinigungen. Eine häufig auftretende, aber verhältnismäßig kleine Bereinigung, die tote Objekte ausschließlich in der neuen Generation findet. Dazu gibt es noch eine volle Speicherbereinigung, die den gesamten Speicher behandelt. *Swiper* ist in etwa genauso schnell wie der *Copy Collector* obwohl nur die Hälfte des Speichers benötigt wird. Die Performance verschlechtert sich nur wenn die meisten jungen Objekte nicht früh sterben.

Unterbrechungen der Applikation zur Bereinigung werden von *Swiper* minimiert, indem die anstehende Arbeit auf mehrere Threads aufgeteilt und gleichzeitig abgearbeitet wird. Je mehr Threads verwendet werden, desto schneller wird die gesamte Speicherbereinigung. Allerdings ist die serielle Speicherbereinigung schneller als die parallele mit nur einem Thread aufgrund von Mehraufwand zur Synchronisierung.

Allokationen werden durch sogenannte thread-local allocation buffers (TLAB) beschleunigt. Diese erlauben der Applikation die meisten Objekte ohne Synchronisierung und ohne Aufruf der Laufzeitmaschine zu allokieren. Die Laufzeit verbessert sich im *binarytrees* Benchmark beinahe um das vierfache. Die meisten anderen Benchmarks verbessern sich auch signifikant.

Abstract

A Tracing Garbage Collector (GC) is a technique for automatic memory management used by many programming languages. The application only allocates new memory, while the GC automatically frees unreachable memory again. In such systems the developer has almost no control when memory is reclaimed. When the GC runs out of memory or when it decides a collection would be worthwhile, the collector stops the application to release memory. It computes the transitive closure of all objects reachable from the application. Then the memory for unreachable objects can be reclaimed. Immediately after discarding the memory, execution of the application can continue.

Dora is a runtime for a statically typed programming language. Functions are lazily compiled to machine code on their first invocation by the baseline compiler.

This work presents the generational GC *Swiper* for the Dora runtime. The collector is based on the *weak generational* hypothesis, that states that most objects die young. Based on this empirical observation the heap is split into *young* and *old* generation. Objects are allocated in the *young* generation and promoted into the *old* one as soon as they become old enough. The GCs duty is to reclaim unused memory. A collection is more effective when applied on memory regions that are more likely to continue garbage. Therefore *Swiper* performs two different kinds of collection: *minor* and *full* collection. Frequent *minor* collections discover garbage solely in the *young* generation, whereas the less common *full* collection handles the full heap. *Swiper* uses different collection schemes for the different collections. *Copy* collection during *minor* collections and *mark-compact* for full collections. *Swiper*'s performance is competitive to pure *Copy* collection, while using half the memory. However, it is less efficient than pure *Copy* collection and *mark-compact* in non-generational workloads.

Pause times are reduced by distributing work to multiple worker threads. Serial collection is faster than parallel collection with one worker thread due to synchronization overhead. Using multiple worker threads reduces pause times compared to serial *minor* and *full* collection.

Swiper also improves allocation performance through the implementation of thread-local allocation buffers (TLAB). This allows the mutator to allocate most objects without synchronization using only a few assembly instructions. The *binarytrees* benchmark becomes almost 4 times as fast with TLAB allocation, most benchmarks improve significantly.

Contents

CHAPTER 1

Introduction

1.1 Tracing Garbage Collection

Programming languages with manual memory management require the developer to manually *allocate* and *free* memory on the heap. Tracing garbage collection (GC) as used in many programming languages on the other hand provides automatic memory management to developers. The application only allocates new memory while the GC automatically frees unreachable memory again if the application runs out of memory. The developer has no way of explicitly releasing memory. Ideally a GC would free all memory that will not be accessed anymore in the future but this problem is undecidable. A GC therefore builds a transitive closure of all objects reachable by the application, hence all memory that could be accessed again. Unreachable memory can be released to the operating system or used for subsequent allocations.

A GC can be tuned for different use-cases, most notably throughput or latency. However GCs might also differ among other properties in heap overhead, allocation performance or in how well they scale with large heaps. Many batch jobs for example are tuned for throughput, the application is supposed to finish as fast as possible while pausing the application for longer periods due to the GC is usually fine. This is in contrast to interactive applications where blocking the application from answering client or user requests for many seconds during a GC pause is unacceptable, lower throughput would be less problematic though.

1.2 The Dora runtime

The Dora runtime[1] executes programs written in the Dora programming language. Dora is open-source and licensed under the MIT-license. Its programming language is statically

[1] `https://github.com/dinfuehr/dora`

typed and requires tracing garbage collection for automatic memory management. The runtime itself is written in Rust, a new programming language focused on performance and safety.

The runtime checks syntax and semantics of the program source code at runtime. It lazily compiles code down to machine code on function granularity right at the first invocation. This implies that Dora uses a compile-only approach and does not execute code in the interpreter until code becomes sufficiently hot. Supporting precise tracing garbage collection in Dora requires cooperation from the JIT-compiler. Dora's JIT supports the generation of both AArch64 [ARM18] and x86-64 [Int19] machine code. The runtime itself is usable on Linux and OS X.

1.3 Motivation

The motivation behind Dora is to serve as a playground for learning and experimentation with respect to runtimes and JIT-compilers. An important component of many runtimes is the garbage collector. Dora aims to have a state-of-the-art GC that reduces pause times and improves throughput by implementing fast allocation. The GC is supposed to be used as default collector and should therefore work relatively well in most use-cases.

At the same time Dora should remain extensible, adding more collectors and switching between them at program startup needs to be possible. In the future it might make sense to add special-purpose GCs that would for example focus on latency. However this is also useful for testing and performance benchmarking.

1.4 Expected result

The goal of this thesis is a general-purpose collector that can be used in Dora as a default collector. The collector is supposed to balance between throughput and latency: pause times should not be too excessive while on the other hand throughput should not be hurt too much. The GC should separate the heap into multiple generations.

The young generation should use a semi-space collector using the Cheney algorithm. This means that objects in the young generation are compacted and the fast bump-pointer allocation can be used instead of the slower free list allocation. In addition compacting objects improves locality and reduces heap size and therefore cache usage.

Since microservices are becoming more and more popular and replacing large monolithic application, there is no special need to tune the GC specifically for extremly large heap sizes for now. The old generation should therefore use *mark-compact*. *Mark-compact* improves object locality compared to *mark-sweep* and allows to use bump-pointer allocation in the old generation as well. Reducing pause times shall be achieved by distributing work onto multiple threads.

The collector needs to cooperate with the compiler to scan the program's root set precisely. The GC needs to be a precise collector to be able to relocate and compact objects without

any restrictions. The compiler emits so-called stack maps at function call sites. This information is used by the collector to find all references in a stack frame when traversing the stack.

1.5 Outline

The rest of this thesis is structured as follows: Chapter 2 covers the required basic knowledge to understand GC implementations. But it is also necessary to understand the trade-offs involved. It covers the basic collection schemes, how generational collection works and concludes with techniques for incremental marking.

Chapter 3 will then discuss state-of-the-art garbage collectors from widely used systems. It presents collectors of three major techniques: generational GCs, collectors using concurrent compaction and non-moving GCs.

After discussing background collectors and other state-of-the-art collectors, Chapter 4 explains the implementation of Dora's generational collector *Swiper*. It covers heap and object layout used in Dora, but also minor and full collection. In particular how generational collection is implemented.

Chapter 5 compares *Swiper* to other collectors available in Dora. Benchmark results are presented and discussed for these GCs. This chapter also compares performance of *Swiper*'s parallel collections relative to the serial implementation in detail.

The thesis is then concluded with chapter 6 that elaborates possible directions for future work with Dora's garbage collectors. It also provides a short summary about the thesis.

CHAPTER 2

Background

Tracing Garbage Collection is a mechanism for automatic memory management utilized by many widely-used programming languages. It increases developer productivity by freeing the engineer from the burden of having to think about the lifetime of dynamic memory on the heap. In languages that use manual memory management the engineer has to *allocate* memory and explicitly *free* that memory again when it is not needed anymore. This is quite error-prone in medium to large applications. One of the problems that might occur is that dynamic memory is freed too early which results in *use-after-free* bugs. Such failures cause undefined behavior and might result in crashes, memory corruption or security issues. A tracing garbage collector (GC) on the other hand provides the illusion of infinite memory - the developer's only responsibility is to allocate memory when needed. Unreachable memory is only freed when either dynamic memory is scarce or the GC deems a collection to be worthwhile.

Ideally a GC would free all dynamic memory that is not going to be used in the future anymore. However this problem is undecidable, a GC therefore only frees memory which was proven to be unreachable from the program. A memory allocation is reachable if it is either directly referenced from a local or global variable. It is also reachable if it is used by another reachable memory allocation. Finding all reachable objects therefore requires to calculate the transitive closure, this process is called *marking*.

Memory allocations in garbage collected systems are often simply called *objects*. The initial set of reachable object is called *root set*. This set contains all objects directly referenced from the current call stack (e.g. local variables or function arguments), CPU hardware registers or global variables. In GC literature the application is often called *mutator* since from the point of view of the collector, the application is mutating the heap.

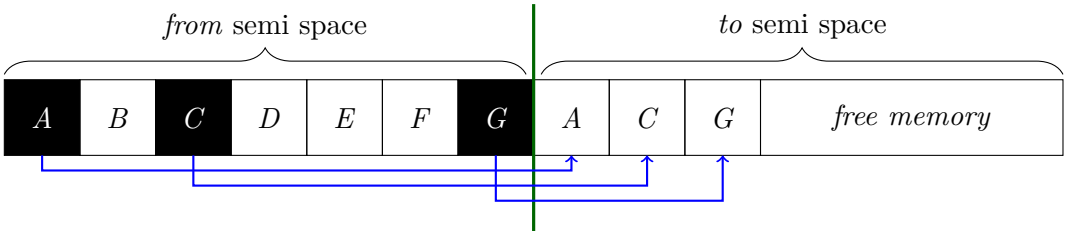

Figure 2.1: Semi Space Collection

2.1 Collection Schemes

This section describes the fundamental collection schemes: *copy* collection (see 2.1.1), *mark-sweep* collection (see 2.1.2) and *mark-compact* collection (see 2.1.3). While these schemes are quite simple, they are still used in similar form in most advanced tracing garbage collectors. This section will try to give an overview about these approaches and discuss inherent trade-offs involved. While *reference counting* is another form of garbage collection, this thesis restricts itself to algorithms used in tracing garbage collection.

2.1.1 Copy Collection

Copy collection is the first collection scheme discussed. A *copy* collector divides the heap into two separate contiguous semi spaces: *from* and *to* semi space. This essentially divides the usable memory into half. Objects are always allocated in the *to* semi space. A *copy* collection is initiated when this semi space becomes full.

At the beginning of a collection *from* and *to* semi space switch roles. Collection then continues by copying objects in the root set from the *from* semi space over to the *to* semi space. When copying an object into *to* space, the original object is marked as copied and the object's new location is written into it (represented by the blue edges in figure 2.1). Starting from the root set *copy* collection traverses all copied objects and copies all reachable objects into the *to* space as well - unless of course the object was already copied. After all reachable objects were copied there will be a single contiguous free memory range at the end of *to*-space. That memory can be used for subsequent allocations by the mutator. All memory in the *from* semi space is considered to be garbage and not used until the next collection.

The main advantage of copy collection is that it only performs work proportional to the size of the live set. This is in contrast to *mark-sweep* and *mark-compact* that will generally need at least one pass over the whole heap. However this becomes less of an advantage the larger the live set size becomes. Especially since *mark-sweep* is dominated by the marking phase which could be run concurrently to the application as well.

Another major upside that shall not be underestimated is that it allows for fast allocation using *bump-pointer* allocation. *Bump-pointer* increments a *top* pointer until it surpasses

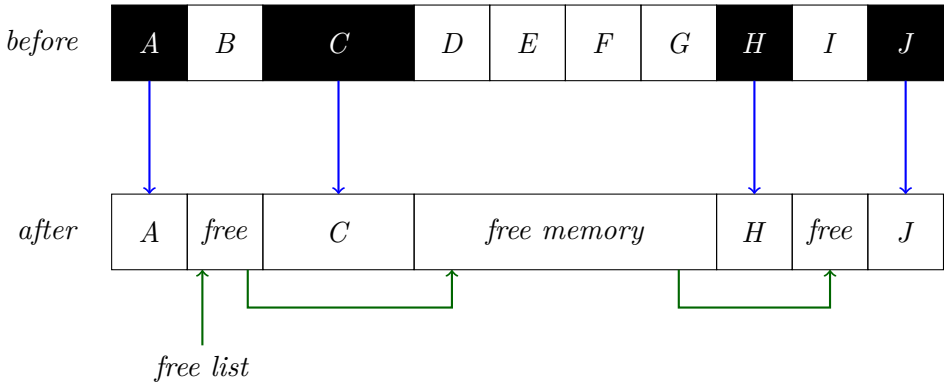

Figure 2.2: Mark Sweep

the *limit* pointer. The memory region between *top* and *limit* is free and can be used for allocations.

During a *copy* collection, objects are copied in object traversal order into *to* space which can be quite effective for data locality. This can be experienced when traversing an array, after the collection individual array elements will generally be allocated next to each other in the order they appear in the array. For *mark-compact* this is much harder to achieve since it has a separate *marking* phase, the version described in section 2.1.3 will keep the order of objects within the heap. *Mark-sweep* does not even relocate objects at all.

The obvious disadvantage of copy collection is that it is only able to use one semi space at a time. This might not be a problem with enough spare virtual memory. While the mutator is running the second semi space is unused and should not increase memory usage. However during a collection this might still use up to twice the memory needed for the same heap size in *mark-sweep* or *mark-compact*.

2.1.2 Mark Sweep

The next collection scheme presented is *mark-sweep*, it consists of two major distinct phases: *mark* and *sweep*. Unlike copy collection *marking* is performed as a separate phase that finds all reachable objects in the heap. Then a subsequent *sweeping* phase scans the heap for unreachable objects and adds those memory regions to a *free list*. Allocations after the collection can be satisfied by finding free memory regions in the *free list*. The key property of *mark-sweep* is that objects are not relocated during a collection. Figure 2.2 illustrates how a *mark-sweep* collection alters the heap.

As a consequence of the distinct *marking* phase, *mark-sweep* needs one extra bit per object to store whether an object is reachable. This bit can either be stored in the object header or in a separate allocated bitmap. During *sweeping* the GC has to scan the whole heap for unreachable objects. While this can be quite expensive, *sweeping* can be broken

up into smaller chunks of work. These chunks are then processed lazily on subsequent allocations. *Sweeping* can also be off-loaded to another thread and run concurrently to the application. Both these approaches are able to improve latency.

In general GCs using *mark-sweep* can have quite short pauses. This is because both *marking* and *sweeping* phases do not require the application to be stopped. Section 2.4 describes incremental and concurrent *marking* that allow reducing pause times.

As a corollary of not relocating objects the heap tends to become more fragmented though. This decreases both data locality and cache usage. In long-running programs fragmentation might also become more of a problem: the collector might encounter situations where there would actually be enough free memory in total available, however not in a single contiguous memory chunk. One approach many garbage collectors use to alleviate such fragmentation is by having a separate space for large objects. Large objects are often allocated in a separate non-contiguous space, the garbage collector keeps track of objects in this space by populating a linked list.

Allocation

Another important distinction to *copy collection* and *mark-compact* is that *mark-sweep* does not use bump-pointer allocation but *free-list* allocation. This means that allocations are served by searching for an appropriate memory cell in the *free list*. There are multiple strategies for finding a cell in the *free list*: *first-fit*, *next-fit* and *best-fit*.

1. *first-fit* simply uses the first memory cell in the *free list* that is at least as large as the requested allocation size. If the cell is larger than required, the cell is split and the unused rest added to the *free list*.

2. *next-fit* is quite similar to *first-fit* but list iteration starts from the position where the previous iteration succeeded. When *next-fit* reaches the end of the *free-list*, iteration continues from the start of the list.

3. *best-fit* selects the memory cell that fits the new allocation the best. This usually implies iterating the *free-list* longer than needed for *first-fit* or *best-fit* but helps to reduce fragmentation.

While *bump-pointer* allocation outperforms *free-list* allocation, this operation is typically only responsible for a small percentage of mutator time. Secondary effects like allocation locality are probably more important for application throughput [BCM04]. With *bump-pointer* allocation objects allocated right after each other are typically stored in consecutive locations and might even be on the same cache line when the objects are small enough. This property generally cannot be uphold with *free-list* allocation, especially when objects are segregated by size.

Segregated-fits Allocation

Many *mark-sweep* collectors segregate objects by size to reduce fragmentation and to speed-up *free-list* allocation. In such a collector the heap typically consists of so-called *blocks*, a *block* has a fixed size (e.g. 8KB) and stores one or more objects of the same size. This means that objects with different sizes are always stored in different *blocks*. Objects that are too large for a single *block* span multiple contiguous *blocks*.

Having many *blocks* with slightly different object sizes would increase external fragmentation substantially, the collector therefore restricts allocation to a set of predetermined size classes. When allocating an object of a given size it is allocated in the smallest size class that is able to fully accomodate the object. Defining the number of size classes and the actual size values has to strike a balance between external and internal fragmentation.

Segregated-fits allocation uses a separate *free-list* for each size class. Allocation therefore simply uses the first entry in the *free-list* for the respective size class. When allocating objects from multiple threads, each thread has its own set of *free-lists* to avoid locking in the common case. Locking is typically only necessary when the thread-local *free-list* is empty and the global *free-list* has to be accessed.

2.1.3 Mark Compact

The last collection scheme to be discussed is *mark-compact*. The major difference to the other collection schemes is that *mark-compact* performs in-place compaction. Similar to *mark-sweep* the first phase is a distinct *marking* phase, the subsequent phases relocate objects and update references. While there are multiple differing approaches for implementing *mark-compact*, this thesis will focus on the Lisp 2-algorithm. In this variant there are three more phases where each phase needs to scan the whole heap:

1. **compute forward address**: This phase scans the heap for live objects and assigns such objects a new address starting from the heap start. This so-called forwarding-address is stored in the object since it is needed by the subsequent phases. This means that objects need an additional word for storing the forwarding address when using *mark-compact*.
2. **update references**: Now that all reachable objects store a forwarding address, this phase scans the heap again for all live objects and updates outgoing references by loading the respective forwarding addresses stored in the referenced objects.
3. **relocate objects**: The last phase moves all reachable objects to the forwarding address by copying the memory byte by byte. Note that old and new object location may overlap and could even be identical if all previous objects were reachable as well.

Figure 2.3 shows how *mark-compact* relocates objects to a consecutive region at the start of the heap. Everything after the relocated objects becomes *free* memory and is

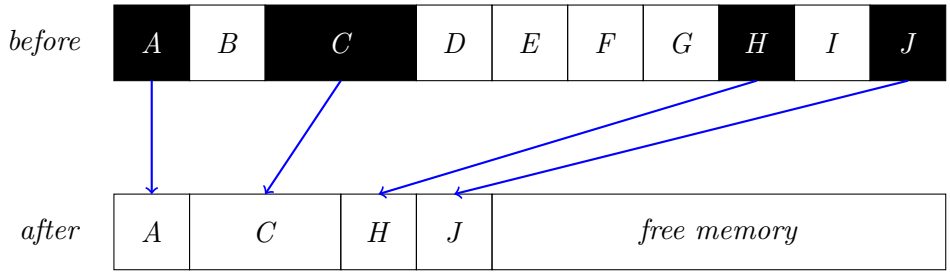

Figure 2.3: Mark Compact

again available for *bump-pointer* allocation. No explicit freeing of memory is necessary. Throughput of *mark-compact* is generally worse compared to *mark-sweep* and *copy collection* because it requires multiple passes over the heap but also the relocation of objects consumes time.

The *mark-compact* algorithm described above keeps allocation order of objects, this can improve locality in many circumstances [BCM04]. It might also be used in a generational GC as a mature space collector. Compacting is beneficial when survival rate is low while access and mutation rates in the mature space are high. In such situations it can improve throughput even though *mark-compact* implies longer collection times.

2.2 Generational Collection

Generational Garbage Collectors [Ung84] are based on the empirical observation of many applications that a large portion of objects die young - the *weak generational hypothesis* [JL96][JHM11]. The GC's job is to reclaim memory and collection is more effective when it focuses on objects or areas with a higher likelihood of discovering garbage. Generational GCs split the heap into multiple generations: e.g. *young* and *old* generation. While GCs can use more than two generations, this thesis will only discuss collectors with at most two generations. Objects are initially allocated in the *young* generation and later promoted into the *old* generation (sometimes called *mature* space) when they become old enough. A generational GC performs two different kinds of collections: *minor* and *major* collections.

During a *minor* collection only garbage in the *young* generation is collected, all objects in the *old* generation are assumed to be alive. *Minor* collections work quite similar to full-heap collections, however a *minor* collection only considers references to *young* objects. So both scanning the *root set* but also visiting the outgoing references of an object will disregard references to other generations. Figure 2.4 exhibits an exemplary heap in a generational GC, blue edges in the graph are traversed during a *minor* collection.

While pointers into the *old* generation can be ignored, a *minor* collection has to take special care of references from the *old* into the *young* generation. This is demonstrated

in figure 2.4 where the young object F is only kept alive by the old object H through the red edge. F would not be reached from the *young* generation.

A naive implementation would scan the complete *old* generation for pointers into the *young* generation on every collection. To avoid this expensive operation the GC tracks all objects in the *old* generation with such references in a *remembered set*. The *remembered set* is kept up-to-date with the help of a *write barrier* that is executed each time a reference is written into the heap like this:

Algorithm 2.1: Write Reference into Heap

```
1 object.field = ref;
```

The barrier adds *object* to the *remembered set* when *object* is an object in the *old* generation and ref an object in the *young* generation. This allows the minor collection to only scan the *remembered set* for *old*-to-*young* references instead of the whole *old* generation. At the same time collections also clean up the *remembered set* from entries that do not reference the *young* generation anymore. Otherwise the *remembered set* might grow until it eventually contains the whole *old* generation again.

Surviving objects of a *minor* collection are either kept in the *young* generation or are promoted into the *old* generation when they have become old enough. The second kind of collection is the *major* collection that collects garbage in all generations. While usually *minor* collections will be more frequent compared to *major* ones, those are still essential to reclaim memory in all generations. *Minor* and *major* collections might even use different collection types, for example copy collection in *minor* collections and mark-sweep or mark-compact for *major* collections.

[Ung84] splits the *young* generation even further (see 2.5): 1. a *new* space to allocate objects in, 2. a *surivor* space for *young* objects that survived the last collection. The *surivor* space is split again into two semi-spaces: *past* and *future* survivor spaces. The *future* survivor space is empty while the mutator is running. Surviving objects in the *new* space and the *past* survivor space are copied into the *future* survivor space during a *minor* collection unless promoted into the *old* generation. After the *minor* collection the two *survivor* spaces switch roles. This layout is based again on the hypothesis that most objects die young, the goal is to increase the size of the usable memory in the *young* generation by limitting the size of the semi spaces.

2.3 Conservative and Precise Collection

Garbage Collectors are classified into either conservative or precise collectors. A precise GC knows exactly what values on the stack, registers and in the heap are valid references into the heap. On the other hand a conservative GC is unable to distinguish heap references from other values for at least one of the locations above.

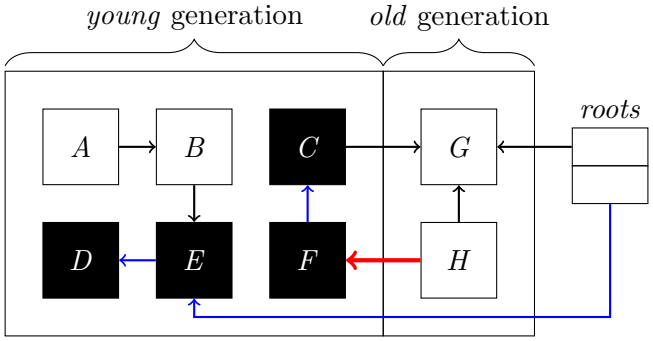

Figure 2.4: Different Reference Kinds in a Generational GC

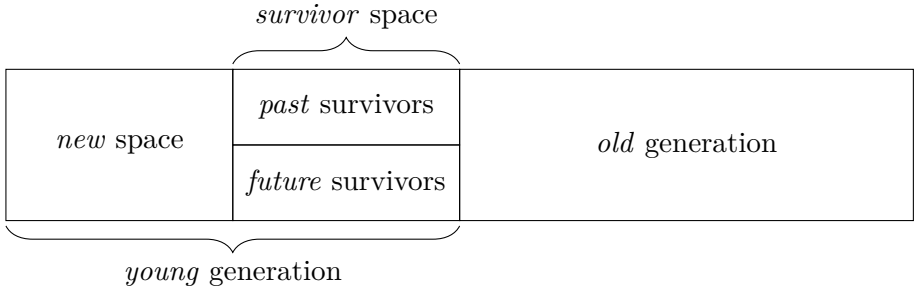

Figure 2.5: Heap Layout in Generational Scavenger by Ungar

Precise GCs either use GC points or tagged pointers to identify heap references on the stack and in registers [Age98]. GC points store which stack entries and registers contain heap references at a single point in compiled code. In a single-threaded environment it is enough to store GC points for each function call site.

Tagging pointers on the other hand allow the GC to infer which values are heap references and therefore does not require additional metadata. The main disadvantage of a precise GC is that it requires cooperation and support from the compiler. Interfacing between managed and native languages also tends to become more complicated since native code is not allowed to have direct references into the heap.

Object maps are typically used to find all references on the heap. Many language implementations store a *type* or *class* pointer with each heap cell to have a simple mean of finding a cell's references. Added to this, the *type* word is also often required to detect an object's size and hence is needed for iterating all objects in a heap.

A conservative GC is simpler to implement but puts additional restrictions on the GC. When a location might reference an object, it has to be kept alive conservatively. This means that a conservative GC keeps more objects alive than a precise GC, however this is often quite minimal [SBM14]. An *ambiguous* reference also adds the restriction that such a referent may not be moved since updating all pointers to it is not allowed.

Conservative GCs that only treat the stack and registers conservatively are therefore still able to relocate most objects.

2.4 Incremental Marking

Marking reachable objects is the most expensive phase of *mark-sweep* and quite costly in *mark-compact*. Fortunately marking can be distributed over multiple smaller pauses to avoid the need for a single long pause to reduce latency. This approach is called *incremental* marking because work is completed in multiple steps.

Incremental marking is best understood with help of the tri-color abstraction [JHM11]. This abstraction assigns each object one of three colors: *white*, *gray* or *black*.

1. **White** objects have not been reached yet and might be unreachable. After marking has finished *white* objects are garbage.
2. While **gray** objects on the other hand are already marked as reachable. However their outgoing references have not been or only partially visited yet. All gray objects have to be visited and turned black before marking can be finished. When transitioning an object from *white* to *gray*, the object is marked reachable and appended to the collector's marking worklist.
3. The **black** color signals that the object itself and all its outgoing references have been visited and therefore marked reachable. A *black* object only references *gray* or *black* objects and is not revisited anymore by the collector. *Black* objects are marked reachable just like *gray* objects. In addition *black* object were already removed from the marking worklist.

Marking finishes when there are no *gray* objects left to visit - the marking worklist becomes empty. All objects are then either marked *white* or *black*: *white* objects are unreachable and therefore garbage, while *black* objects need to survive the collection. During marking the set of *gray* objects is often called wavefront in GC literature. This is because conceptually *gray objects* are the boundary between *white* and *black* objects. Transitioning an object from *white* to *gray* adds to the wavefront, while marking an object *black* advances the wavefront. Some implementations also retreat the wavefront by reverting the color of an object from *black* to *gray*.

2.4.1 Weak and Strong Tri-Color Invariant

When both mutator and collector progress independently from each other, it could happen that the mutator changes the object graph in such a way that the collector cannot find all reachable objects. The mutator could install a reference to a *white* object A in a *black* object B and then remove every other path to A. The collector does not revisit B because it was already marked *black*. Since there is no other path to A as well, A

would be falsely considered garbage by the collector. The collector can only fail to mark reachable objects when both of these two conditions hold:

1. the mutators installs a reference to a *white* object in a *black* object and
2. no path from some *gray* object via possibly other *white* objects to the *white* object exists.

The *weak* and *strong* tri-color invariants describe properties that must hold during incremental marking. If either one of them holds, the collector is guaranteed to find all reachable objects with incremental marking. This is achieved simply by preventing one of the conditions above.

The *strong* tri-color invariant states that there are no pointers from *black* objects to *white* objects in the object graph. When the mutator stores a pointer in an object, it needs to check whether this would create a *black*-to-*white* pointer and then handle this situation appropriately.

The *weak* tri-color invariant allows *black*-to-*white* pointers but demands that all *white* objects installed can be reached through some path from a *gray* object. The *gray* object either needs to point to the *white* object directly or indirectly through other *white* objects. *Black* objects on the path are not allowed since the collector would stop marking at these objects.

2.4.2 Mutator Color

In the tri-color abstraction each heap object is assigned a color based on its state in the marking process. Conceptually the mutator's roots get a color as well: either *gray* or *black*. A *gray* mutator can have references to *white*, *gray* and *black* objects. It either means that the roots have not been scanned already or that they have to be scanned again before marking can finish.

On the other hand a *black* mutator is only allowed to contain pointers to *gray* or *black* objects. It differs to a *gray* mutator when finishing marking: with a *black* mutator the stack does not have to be rescanned before marking finishes. When using incremental marking with a *gray* mutator, roots need to be rescanned since there might be more pointers to *white* objects. In the worst case such an additional *white* object might make a large subgraph in the heap reachable and increase pause time significantly.

Mutator color also influences the initial color of allocated objects. Allocating objects *black* might prolong the lifetime of new objects compared to *white* allocation. When allocating objects *black* during a collection, they are guaranteed to survive the current collection cycle even if they are not used anymore. *Gray* mutators can use *white* allocation, while *black* mutators need to allocate objects *black*.

2.4.3 Incremental Update and Snapshot-at-the-Beginning

Solutions for incremental marking are classified into either *incremental update* or *snapshot-at-the-beginning* [Wil92]. *Snapshot-at-the-beginning* implementations retain all objects that were live when the collection was started. Such algorithms break the *strong* tri-color invariant: references from *black* to *white* objects are temporarily allowed during a collection. Nevertheless the *weak* tri-color invariant is still fulfilled and therefore those objects are eventually reached by the collector. Snapshot collectors also require a *black* mutator and hence non-*white* allocation. If this would not be the case, the mutator could insert the sole pointer to a *white* object behind the wavefront and then drop the reference.

An *incremental update* implementation is called this way because the mutator notifies the collector on updates to the object graph. It preserves the *strong* tri-color invariant by preventing the creation of *black*-to-*white*-references. While *snapshotting* retains all objects that were live at the start of the collection, *incremental update* tries to only retain objects that are live at the end of a collection. Objects that become unreachable before they are marked are actually detected as not reachable by the collector. Allocating objects *white* is permitted.

2.4.4 Barriers

This section will give an overview about *barriers*, that are used to coordinate the mutator with the collector. While this thesis does not discuss all possible barriers [Pir98], it covers those relevant for it. A *barrier* executes additional code when reading from or writing into the heap. Based on the type of heap operation, barriers are distinguished as *read* and *write* barriers. Barriers will also be grouped based on whether they are used with a *gray* or *black* mutator.

Gray Mutator Barriers

Barriers in this section are used with a *gray* mutator: the *Steele* and *Dijkstra* barrier. Both are *write* barriers and are used in *incremental update* algorithms. Therefore they abide by the *strong* tri-color invariant.

The *Steele* barrier [Ste75] prevents *black*-to-*white*-references by executing 2.2 on writes into the heap. The *write* barrier checks whether the source object is *black* and the referent *white*. If this is the case the color of the source object is *reverted* from *black* to *gray* and the object later rescanned by the collector. The *Steele* barrier therefore is said to retreat the wavefront because it creates more work for the collector.

The *Dijkstra* barrier [DLM$^+$78] uses a slightly different approach but still intercepts writes into *black* objects (see 2.3). When a reference is written into a *black* object, the barrier marks the referent reachable as well. While *Steele* retreats the wavefront, *Dijkstra* adds to it.

Algorithm 2.2: Steele barrier

```
1 object.field = ref;
2 if color(object) = black then
3 |   if color(ref) = white then
4 |   |   revert(object);
5 |   end
6 end
```

Algorithm 2.3: Dijkstra barrier

```
1 object.field = ref;
2 if color(object) = black then
3 |   mark(ref);
4 end
```

Black Mutator Barriers

The *Baker* and *Yuasa* barriers are used with *black* mutators. The *Baker* barrier [Bak78] is the only *read* barrier described and implements an *incremental update* solution. It ensures that every reference read from the heap is non-*white*. Listing 2.4 marks references read from a *gray* object reachable to guarantee this. Since all references on the stack and read from the heap are either *gray* or *black*, installing a reference to a *white* object into a *black* object is impossible. The barrier only needs to check for color *gray*, *black* objects only reference non-*white* objects already. While *white* objects cannot be encountered because of the *black* mutator.

Algorithm 2.4: Baker read barrier

```
1 ref = object.field;
2 if color(object) = gray then
3 |   mark(ref);
4 end
5 return ref;
```

The *Yuasa* barrier [Yua90] allows for a *black mutator* algorithm with just a *write* barrier. As a *snapshot-at-the-beginning* barrier, it keeps all objects alive that were reachable at the start of the collection. It works by marking the overwritten value as reachable as can be seen in listing 2.5. Inserting a reference to a *white* object into a *black* object is temporarily allowed. Therefore this barrier only adheres to the *weak* tri-color invariant. The *Yuasa* barrier conservatively assumes that an overwritten value was hidden behind the wavefront and therefore has to be marked.

Algorithm 2.5: Yuasa barrier

```
1 mark(object.field);
2 object.field = ref;
```

Concurrent Marking

While the barriers described are used in incremental marking they can also be used in a similar form for concurrent marking. Concurrent marking marks reachable objects while the application keeps running. Compared to incremental marking special care has to be taken about the ordering of memory operations used in a barrier. In addition reads and writes to the heap are required to be atomic.

CHAPTER 3

Related Work

Tracing GC is used for automatic memory management in many widely used systems. This chapter discusses the garbage collectors of some of those systems, in particular how the heap is organized and how a collection operates. It also tries to give a brief glimpse into tradeoffs the different implementations had to make. The GCs are broadly classified into 1. generational, 2. concurrent compacting and 3. non-moving collectors. First the generational garbage collector V8 (3.1) is introduced. With Shenandoah (3.2) and ZGC (3.3) two collectors are then discussed that compact objects concurrently to the application. Finally the non-moving collectors JavaScriptCore (3.4) and Go (3.5) are presented.

3.1 V8

V8 is a JavaScript engine used in the Chromium project, it uses a generational collector named Orinoco. The heap is split into a *new* and *old* space. Objects are allocated in the *new* space and later promoted into the *old* space when they survive enough collections.

While usually most objects allocated by the mutator are stored in either of these spaces, there are more spaces for special object kinds: Executable code objects are allocated in the *code* space, while large objects are part of the *large object* space. Immutable and immortal objects are stored in the special *read-only* space and *map* objects are allocated in the separate *map* space.

*Map*s in V8 are an important performance optimization used in many dynamically typed languages [CUL89]. Strictly speaking a JavaScript object is actually a key-value dictionary, hence each field access would require an expensive dictionary lookup. However in practice many objects share the same object layout, *maps* exploit this property by allowing cheap field access through the field's offset in the object similar to statically-typed languages. A *map* stores information shared by all objects of this map kind, including

field names, field offsets and object size. Each object stores a reference to its *map* object in its first word, Orinoco uses this word to lookup an object's size and pointer fields.

Each space consists of multiple pages, note that page in this context is a V8 term and shall not be confused with an OS page. A V8 page is a contiguous memory area that is always aligned to 512K and usually 512K in size. Each page consists of a header and the object area, the header stores the page metadata while the object area stores the actual heap objects. Aligning pages to 512K makes it cheap to calculate the page address and hence allows fast access to the page's metadata with a simple bitmask from the object address. As already mentioned, pages are 512K bytes large with the exception of pages in the *large object* space. Each *large object* is stored as a separate page, therefore the page has to be large enough to accomodate the respective object.

3.1.1 Minor Collection

A minor collection focuses collection of garbage in the *new* space, objects in all other spaces are assumed to be reachable and are not collected. *New* space is further split into two semi spaces (*from* and *to* space) that are collected using a parallel scavenger similar to the one described in [Hal84]. Initially the *to* space is empty, all surviving objects in the *from* space are copied to the *to* space or promoted into the *old* space if the object is old enough.

V8 tracks references from other spaces into the *new* space to achieve faster collection times. All objects in non-*new* spaces are assumed to be alive and therefore references from that space to the *new* space keep additional objects in the *new* space alive. If V8 had not remembered these references, the whole heap would have to be scanned to find all references into the *new* space.

V8 achieves this by executing a generational write barrier anytime a reference is stored in an object. The barrier checks if a reference to a *new* object is stored in a non-*new* object. If this is the case the field's address is appended to the store buffer.

When the store buffer becomes full, all its entries are transfered to the old-to-new remembered set. The remembered set stores all interesting slots in a V8 page as a bitmap. If a bit is set, the corresponding slot is considered part of the remembered set. Allocating a bitmap for the whole page even in cases when only few bits are set is quite expensive, hence a page's bitmap is divided into multiple individual buckets. A bucket is a fixed size section of the original bitmap and is only allocated when at least one bit in the respective area is set.

Processing the store buffer can be done concurrently to the application while execution of the application continues with a second alternative empty store buffer. Typically when one store buffer becomes full, the other buffer is already processed and the application can switch the buffers immediately without waiting.

Note again that V8 keeps track of the exact slot for such old-to-new references. While Dora's generational garbage collector on the other hand marks small memory regions

(so-called *cards*) *dirty* instead of individual slots.

One of the defining characteristics of a generational GC is that objects are eventually promoted into the *old* space when they become old enough. In V8 all objects that survive a second collection are promoted into old space. V8 uses an *age marker* to determine the number of collections an object has survived. After a minor collection the address of the end of the *new* space is stored in the *age marker*. All objects located before this marker have already survived a minor collection, they are going to be promoted on the next collection. While objects that were allocated after the last collection are located after the marker and hence stay in *new* space for one more collection.

3.1.2 Full Collection

In contrast to the *minor collection*, the *full collection* collects garbage in all spaces. A full collection consists of three major phases: 1. marking, 2. evacuation and 3. pointer updates.

When starting a full collection the GC decides which pages in the *old* or *code* space are evacuated during the collection, all such pages are marked as an evacuation candidate. All pages in the *new* space are treated as evacuation candidates as well. Evacuating a page moves all live objects to another page, therefore V8 prioritizes pages for evacuation with the least live bytes. Since V8 has to select the evacuation candidates before marking starts, liveness information is based on the last collection.

Marking in Orinoco as usually builds the transitive closure of all reachable objects in the heap. While traversing reachable objects, the GC also needs to record slots with references from an object on a non-evacuated page to an object that will be evacuated. All these slots are collected in the old-to-old remembered set, which is handled similarly to the old-to-new remembered set used for minor collections. V8 supports both incremental and concurrent marking to move at least part of the work out of the GC pause to reduce yank, this is described in more detail in the next section (3.1.3).

In the second phase pages chosen to be part of the collection set are evacuated into newly allocated and empty pages. Moreover the GC installs a forwarding pointer to the object's new location in each relocated object. After copying the object, V8 also needs to scan the copied object for interesting references. References into the young generation are added into the old-to-new remembered set, while references to evacuated pages are added to the old-to-old remembered set.

In the last phase the GC needs to update all references to relocated objects in all pages except for those that already got evacuated. V8 can use the old-to-old remembered set to find all those references within a page and updates them by following the forwarding pointer stored in the relocated object. This phase also updates the old-to-new remembered set by removing slots that do not store a reference to the *new* space anymore. This could either be because the target object was promoted into *old* space or the slot was updated by the mutator to reference an object in *old* space.

3.1.3 Incremental and Concurrent Marking

The basic idea behind incremental marking is that it divides marking work over multiple smaller GC pauses to avoid having one large GC pause on the main thread. When V8 is embedded within the *Chromium* project, incremental marking steps can even be scheduled when the application is idling [DEE+16]. Concurrent marking runs on one or more worker threads while the application continues to run. A Dijkstra style write barrier [DLM+78] is used to prevent the mutator from hiding objects from the collector by ensuring the strong tri-color invariant holds [HJH10].

Algorithm 3.1: V8 incremental write barrier

```
1 object.field = value;
2 if color(value) = white then
3 |   mark(value);
4 end
```

The write barrier does not check whether the *object*'s color is actually black to avoid the need for an expensive memory fence between writing into an object's field and checking its color. This certainly leads to a more coarse-grained write barrier that detects more objects reachable and in consequence more floating garbage.

An additional complication with concurrent marking for dynamically typed languages like JavaScript is that some actions can cause object layout changes, e.g. adding a new property to an object. At the same time V8 stores both tagged and untagged values in an object for performance reasons, which means that the value alone is not sufficient to determine whether the given value is an object reference. The object's *map* has to be inspected as well, it knows which fields might contain object references.

With active concurrent marking it could now happen that while a concurrent worker thread marks an object, the application changes the object's layout concurrently. The GC could therefore already read an untagged value but still assumes it is a valid pointer according to the old layout. Such situations need to be noticed by the GC, this is the reason objects are marked black and immediately traced when its layout is changed by the application thread. When an object is marked in a concurrent marking thread, V8 first reads all references in an object into a snapshot buffer. Afterwards it tries to mark the object black atomically, if this operation succeeds the object layout has not changed, all references in the snapshot buffer are valid. If marking the object black has failed though, the object's layout was changed by the main thread and therefore references in the snapshot buffer might be unsafe. In this case the snapshot's content is ignored, no other work is required.

3.2 Shenandoah

Shenandoah [FKD+16] is a GC for OpenJDK that focuses on minimizing latency. It achieves this by evacuating objects while the application is running.

Shenandoah is not a generational garbage collector like V8's Orinoco 3.1. Generational garbage collectors often perform a minor collection in a stop-the-world pause and even though the young generation is likely to be fast, there can still be latency spikes which Shenandoah tries to avoid.

Shenandoah organizes the heap as a set of equally-sized regions. Objects are allocated sequentially within a region and apart from *humongous* objects do not span multiple regions. *Humongous* objects span at least one or more consecutive regions.

A garbage collection cycle in Shenandoah is organized in multiple phases:

1. **Start Marking**: Shenandoah marks objects in the rootset during a stop-the-world.
2. **Concurrent Marking**: GC worker threads mark reachable objects concurrently to the application threads. Shenandoah uses a Yuasa write barrier [Yua90] that marks overwritten values reachable.
3. **Finish Marking**: This phase finishes marking by draining all Snapshot-at-the-beginning buffers. Subsequently a collection set is chosen based on the amount of live objects or bytes in each region. Basically the collection set is a set of regions that gets evacuated during the collection. Objects in other regions are not relocated. Finally objects in the root set that are part of the collection set are evacuated.
4. **Concurrent Evacuation**: Walk all regions in the collection set and evacuate objects unless they were already evacuated by the mutator.
5. **Start updating references**: Makes the heap parsable for the next concurrent *update references* phase. The application is paused in this phase.
6. **Concurrent updating references**: Traverse the heap and forward references concurrently to the application.
7. **Finish updating references**: Pause the application again, there are no more references into the collection set left. The regions in the collection set can now be freed.

Shenandoah does not get rid of stop-the-world pauses completely but limits work done in those pauses rigorously. Required pauses only have work proportional to the size of the root set but not the live-set or heap size. Updating references can also be combined with the next collection's marking phase if the GC decides so. In this case the pause after concurrent evacuation finishes evacuation and the current collection cycle.

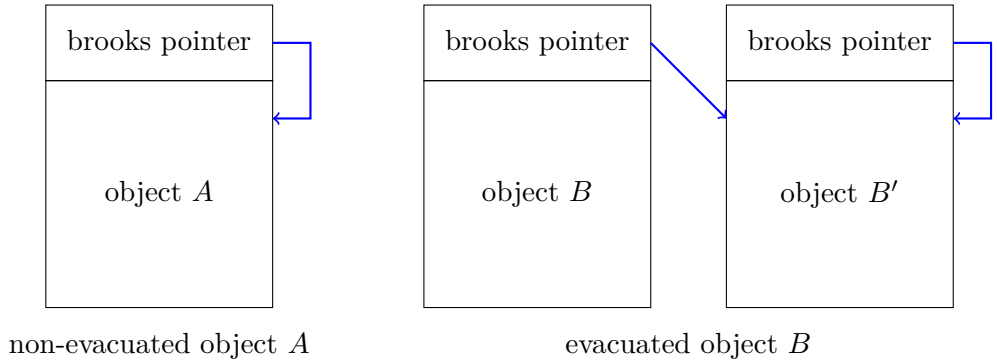

Figure 3.1: Brooks pointer

3.2.1 Concurrent Evacuation

Shenandoah stores a Brooks pointer [Bro84] right before each individual object. Although this increases memory usage it is necessary for evacuating objects concurrently. The Brooks pointer references the evacuated version of this object, if there is no evacuated version it references itself (see figure 3.1).

Shenandoah requires both a read and write barrier for heap accesses to allow for concurrent evacuation. Its read barrier simply follows the brooks pointer before loading a field as can be seen in listing 3.2.

Algorithm 3.2: Shenandoah read barrier

```
1 realObject = object.brooks;
2 value = realObject.field;
```

Even though the read barrier only requires one additional instruction, it can still degrade performance considerably. Read barriers are quite performance sensitive since they tend to be so frequent - about an order of magnitude more often than a write barrier [YBFH12].

Shenandoah already requires a Yuasa write barrier for concurrent marking. In addition Shenandoah requires a second write barrier while objects are evacuated. Fortunately write barriers tend to be less frequent than read barriers, this allows Shenandoah to afford more complex logic in them. When concurrent evacuation is running, writes into objects that are in the collection set are intercepted. Before such a write the object is first evacuated and the write is performed later outside of the collection set on the copied object. Therefore no writes into the collection set can happen. When writing a reference into the heap, the write barrier also implements a read barrier for the referent. This stops the distribution of unforwarded references over the heap. Also note that both read and write barriers need to be performed for both reference and non-reference types.

color	good color	bad mask
marked0	0001	1110
marked1	0010	1101
remapped	0100	1011
finalizable	1000	0111

Table 3.1: Colors in ZGC

Evacuating an object copies it into its new location. After copying the GC tries to update the Brooks indirection pointer with an atomic compare-and-swap operation. When this operation succeeds the evacuation was successful, otherwise it was beaten by another application or GC worker thread that evacuated the object first. The first thread that updates the indirection pointer wins, the other threads either unroll the allocation or fill the allocation with a filler object if this is not possible.

3.2.2 Traversal GC

Shenandoah supports another mode of operation that merges marking and evacuation of objects into a single phase. The collection set is chosen in the *start marking* pause based on the liveness information of the last collection. The first collection after starting the program does not have liveness information however. Marking an object and possibly evacuation, if the object is part of the collection set, is performed at once. There are only tree phases with Shenandoah's Traversal GC: 1. Start Traversal, 2. Concurrent Marking and Evacuation, 3. Finish Traversal.

3.3 ZGC

ZGC is another GC that tries to minimize GC pauses for OpenJDK [Lid18]. Nevertheless it is different and interesting enough to warrant its own section in this thesis. Similarly to Shenandoah 3.2 it uses a read barrier, does not perform generational garbage collection and separates its heap into regions. However in ZGC parlance regions are called pages. There are small, medium and large pages, which means not every page has the same size, but each page is a multiple of 2MB on x86-64. Large objects are stored alone in a single page, while small and medium pages are able to store multiple objects.

3.3.1 Colored Pointers

ZGC uses colored pointers or pointer tagging to store additional metadata in references. The color of the pointer can be used to determine whether the object was already marked or relocated. There are four different colors: *marked0*, *marked1*, *remapped* and *finalizable*. Each so-called color is actually an individual bit in the pointer.

ZGC's read barrier (listing 3.3) checks whether the pointer has the current *good* color, which is the color that the active GC phase requires. If the pointer does not have the

required color, the barrier executes a slow path that *repairs* the pointer by e.g. marking or relocating the object. Afterwards execution is resumed with the pointer updated to the *good* color.

Table 3.1 lists the *good* colors and the corresponding *bad* masks in ZGC. The *bad* mask is the negation of the *good* color and the value that is actually used in the *read* barrier. The pointer is tested against the bad mask, this prevents entering the slow path for *null* references.

Algorithm 3.3: ZGC read barrier

```
1 if object.field & badMask then
2 |   slowPath(object, field);
3 end
4 value = object.field;
```

ZGC gets by with just a read barrier - no writer barrier required. Compared to Shenandoah, read barriers are only required when loading references from the heap but not for non-reference types like integer or floating point. On the other hand the read barrier invokes the slow path solely based on the color of the pointer - not on the state of the object. This means that application threads might have to call into the slow path multiple times when there are multiple pointers to the same object. Which is not a problem for correctness since the slow path can simply check whether the object was already marked or relocated, but it could decrease throughput by visiting the slow path more often. Also ZGC requires to *repair* each pointer during the relocating phase through its read barrier even though the object is not part of the collection set. Shenandoah can here be a bit more restrictive with its dedicated write barrier.

Pointers in the root set are *repaired* during the stop-the-world pause. Intuitively this means that every local variable is guaranteed to have the right color. Additional references can only come from reading from the heap, which again is protected by a read barrier. That implies that pointers on the stack are always of the right color. Another nice property of ZGC's read barrier is that it could potentially enable more use cases like storing infrequently accessed pages on the disk and only load it back when it is accessed again.

Depending on the instruction-set architecture colored pointers are implemented differently: On AArch64 the highest byte of a 64-bit word is automatically masked out on every store and load. The color bits can therefore be stored in this byte without any consequences.

This is not the case on x86-64 where masking the color out of the pointer would have to be done manually before each load and store. ZGC avoids these additional masking instructions by mapping the heap multiple times into the virtual memory: for *marked0*, *marked1* and *remapped*. Depending on the GC phase loads and stores on the same object happen on different virtual addresses, as listed in table 3.2. On this architecture color bits are stored in bits 42-45 (figure 3.2). The low 42 bits store the object offset, which

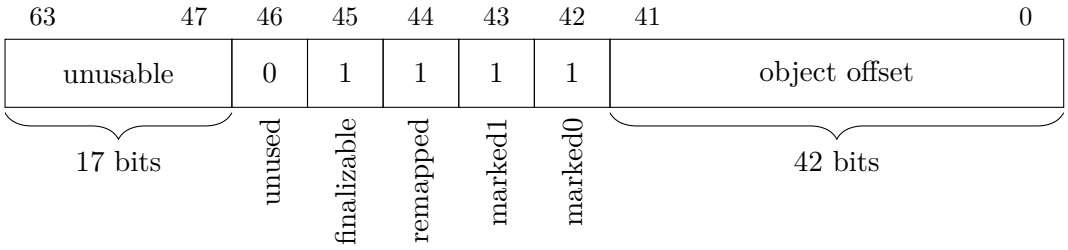

Figure 3.2: Encoding metadata in pointers on x86-64

color	bits	virtual address range
marked0	0001	4-8TB
marked1	0010	8-12TB
unused	0011	12-16TB
remapped	0100	16-20TB

Table 3.2: Virtual address space in ZGC on x86-64

limits the heap to 4TB in size with ZGC. While the highest 17-bits and the unused bit 46 are always zero.

3.3.2 Collection Overview

The phases in a collection cycle are again very similar to Shenandoah: 1. Start Marking Pause, 2. Concurrent Marking, 3. Finish Marking Pause, 4. Concurrent Prepare Relocation, 5. Start Relocation Pause, 6. Concurrent Relocation.

When ZGC decides to start a collection the application is initially paused to start marking. In the pause all objects directly referenced from the root set are marked. While doing this all pointers in the root set are updated to have the same *marking* color, either *marked0* or *marked1*. ZGC will use *marked0* as marking bit when the last collection used *marked1* and vice-versa. This phase does not only perform marking but also forwards pointers to the object's new location if needed since there is no separate *update references* phase in ZGC. ZGC calls that *remapping* since references might have to be forwarded but no relocation has to be performed. References with the last collection's marking color need to be both forwarded and marked, while those with the *remapped* color only need to be marked.

Concurrent marking then continues this work by marking all objects reachable from the root set in GC worker threads. ZGC's read barrier calls the slow path for unmarked references in the application threads. Such references are pushed into a thread-local marking buffer and are only passed to the worker threads when the buffer becomes full.

When the marking threads finish their work the application is paused again to finish marking. All marking buffers are drained and all unmarked references left are processed. If

the final marking phase takes too long ZGC can stop the pause and repeat the concurrent marking phase.

When marking is finished, relocation is prepared. This entails processing non-strong references and choosing a collection set concurrently to the application. As soon as a collection set is chosen the application can be paused again to start the relocation phase. Objects directly referenced by the root set are relocated and updated to the new good color *remapped*. The new location of relocated objects is stored in a per-page *forwarding table* outside of the page itself. It is organized as a key-value dictionary, which maps the object's old offset in the page to the new memory address.

Afterwards the GC worker threads relocate objects by sequentially visiting all marked objects in the pages of the collection set. Just like in Shenandoah, application threads might compete with worker threads to relocate the object. The first thread to update the *forwarding table* atomically wins. Again ZGC's read barrier detects references that are not *remapped* yet and will remap and/or relocate references loaded from the heap. There is no additional pause when the relocation phase ends, the GC cycle stops when all worker threads have finished their work.

As already discussed ZGC does not have a separate *update references* phase, so the next collection's marking phase might still have to forward references. This means the forwarding table is needed until the next collection's marking phase finishes. Shenandoah stores the forwarding pointer (the Brooks pointer) right before each object while ZGC stores this information outside of the heap in the already mentioned *forwarding table*. This allows ZGC to immediately free a page when all objects in it are relocated, only the *forwarding table* needs to be kept around for *remapping*.

3.3.3 Heap Layout

Another interesting feature of ZGC is that it distinguishes physical and virtual memory for the heap. ZGC allocates a physical memory backing storage that can be expanded up to the maximum heap size. It reserves a virtual memory address space of currently 4TB on x86-64 as well. When allocating memory in the heap, physical memory is mapped to a contiguous free address space in the virtual memory. The physical memory for an allocation does not have to be contiguous, only the virtual address area needs to be. Usually there is plenty of virtual memory (4TB) while physical memory is more scarce, so this requirement should be easy to satisfy. This mechanism lets ZGC suffer less from fragmentation. Usually an allocator would fail to allocate x bytes of memory when not available as a single contiguous memory chunk - even though there would be enough free memory in total. ZGC can map multiple free chunks of memory in the physical memory to a single contiguous virtual memory address range to satisfy the allocation.

3.4 JavaScriptCore

JavaScriptCore (JSC) is WebKit's JavaScript engine, its GC is completely non-moving, this means objects are never relocated [Piz17]. Another defining feature of JavaScriptCore is that it scans the root set conservatively. This means that for determining the root set all words on the stack and in registers are inspected for pointers to objects. While JSC is conservative in the root set, it is still precise for references on the heap. JSC knows which other objects an object references.

3.4.1 Heap Layout

Not relocating objects also entails a different heap organization compared to many moving GCs. The heap is organized as a set of blocks of 16KB size. Objects are segregated by size, that means each block only stores objects of the same size. Objects larger than approximately 8KB are simply allocated with the native memory allocator using *malloc*. Large objects need to be tracked separately in a linked list and can be distinguished cheaply from objects in blocks by their alignment: Objects in a block are aligned to 16 bytes, while large objects are only 8 byte aligned.

3.4.2 Collector Phases

A collection cycle consists of two major phases: *marking* and *sweeping*. Marking is mostly done concurrently to the application, while blocks are usually swept lazily on subsequent allocations in the allocation slow path. Sweeping adds unreachable objects in the block to a free list, subsequent allocations can use the memory from the free list again.

JSC supports adding of so called user-defined constraints. These constraints are executed during a pause and can perform additional actions and mark more objects. Even gathering and marking the root set is implemented as such a constraint. Constraints are executed until no more objects are marked - hence a fixpoint is reached. However, if there were objects marked a concurrent marking phase has to follow. When concurrent marking finishes, constraints need to be run again. A GC cycle might have multiple transitions between concurrent marking and constraint execution.

JSC also supports generational garbage collection and is able to perform either eden or full collections. Both collection types essentially work the same, eden collections just do not clear the mark bits when starting a new GC cycle. The collector can decide to perform an eden instead of a full collection when there is enough spare memory. Letting mark bits stick means that only objects allocated since the last GC cycle are not marked yet, so only recently allocated objects are marked - the eden generation. The advantage of this approach is that eden collections do not only work very similar to full collections but can also be executed mostly concurrent to the application. Typical generational GCs usually perform minor collections in a pause which could increase latency.

3.4.3 Retreating Wavefront

JSC combines both generational and concurrent marking barrier into a single write barrier quite similar to the Steele barrier [Ste75]:

Algorithm 3.4: JSC write barrier

```
1 object.field = value;
2 if color(object) = black then
3 |   slowPath(object);
4 end
```

Requiring only one barrier improves throughput compared to having two distinct barriers. The *Steele* barrier reverts the color of black objects on updates back to gray and revisits the object in marking. In the slow path of the write barrier the object is added to the remembered set, all objects in the set are later revisited by the collector. This solves the problem of storing a white object in a black object and hence hiding it from the collector.

The same barrier is also used as a generational barrier where the meaning of the *color* of objects changes: *black* means *old*, *gray* is *remembered* and *white* is *eden*. JSC's barrier is used as a generational barrier outside of a collection cycle and collects old objects with potential references to eden objects in the remembered set. An eden collection treats the remembered set as part of the root set, while a full collection does not require this information and just ignores this information.

Compared to the original Steele barrier JSC does not check if the color of *value* is indeed *white* in 3.4. The barrier is therefore less precise and adds more objects to the remembered set than actually necessary. For correctness it is necessary that the store in line 1 is executed before loading the color of the object. On modern CPUs an expensive memory fence is necessary to guarantee this, therefore JSC makes the fence conditional and enables it only while a collection cycle is running. Algorithm 3.5 shows how the write barrier is actually implemented in JSC. *barrierThreshold* is a global variable that is updated when starting and finishing a collection cycle. This variable ensures that the slow path is always executed during a collection cycle. Outside of a collection the slow path is only invoked for *black* (respectively *old*) objects.

Algorithm 3.5: JSC write barrier

```
1 object.field = value;
2 if object.color ≤ barrierThreshold then
3 |   slowPath();
4 end
```

The slow path of the write barrier looks similar to 3.6. Line 1 checks if a collection is currently active. During a GC cycle the memory fence on line 2 stops the CPU from reordering the store to the field with the load of the object's color on line 3. As already

mentioned during concurrent marking the barrier's slow path is always executed. The slow path therefore checks whether the color of the object is actually black and bails out if not. The object is then appended to the remembered set for both the generational and concurrent barrier.

Although executing the slow path for every write barrier certainly hurts throughput, this solution avoids executing an expensive memory fence outside of collections.

Algorithm 3.6: write barrier slow path

```
1 if collecting then
2 |   fence();
3 |   if color(object) ≠ black then
4 |   |   return
5 |   end
6 end
7 addToRememberedSet(object);
```

3.5 Go

Go is another programming language that uses tracing garbage collection for memory management. Its GC is a concurrent, non-moving and non-generational GC that was optimized for very low latency [Hud18]. Go uses a *mark-sweep* collector, it stops-the-world for starting and terminating marking. Sweeping is performed after the pause for mark termination concurrently to the application, but also lazily on subsequent allocations.

3.5.1 Heap Layout

Similar to JSC 3.4 objects are segregated by size in the heap, however Go organizes this differently and does not use the system's memory allocator. Go's GC design was probably also influenced by the design of the Go programming language were interior pointers are quite common. Allowing interior pointer means that pointers are able to point into the middle, not just to the start of an object. In a size-segregated heap it is cheap to determine the object start address from a random interior pointer. Even though an interior pointer might only reference a small part of the object, it keeps the full object reachable.

Go divides the full address space into so-called *arenas* with a fixed-size of 64MB. Within an *arena* Go allocates *spans* which themselves then store the actual objects. *Spans* are always sized to be a multiple of 8KB and only store objects of the same size class. Small objects up to 32KB are separated into almost 70 size classes. Large objects are allocated in its own *span*, such a *span* might even span multiple contiguous *arenas* to accomodate very large objects.

Another important difference to many other platforms is that Go does not store a *type* pointer in each object's first word. Usually this word is used to identify an object's size and what words in it store references to other objects. Go instead allocates a bitmap for each arena that stores which words contain a heap reference. The bitmap stores exactly two bits for each word in the arena: 1. the bottom bit, which is set if this word stores a heap reference and 2. the top bit, which is set when there are more pointers in the subsequent words in the current object, otherwise scanning the object can be finished since it does not contain more heap references.

While the *arena* owns the bitmap to detect heap references, the *span* stores both the allocation and marking bitmap. The *allocation* bitmap stores what slots in the *span* are currently used, while the *marking* bitmap is used during a GC cycle to store what slots are reachable. After a collection unreachable slots are free, therefore the *allocation* bitmap can be replaced by the *marking* bitmap. The *marking* bitmap is then replaced with new zeroed memory. Go only needs one bit for each slot in the *span*: either the object is marked or unmarked. Gray objects in Go are both marked and on a marking queue, while a black object is marked but not in any marking queue anymore.

Allocations in Go do not require locking in the fast-path. Each goroutine has a local cache of *span*s that can be used for allocation. The allocator requests a *span* from the cache for the given size class. The *span* knows the index of its next free slot, the allocator then sets the corresponding index in the *allocation* bitmap.

3.5.2 Concurrent Marking

Obviously Go also marks reachable objects without stopping the application to reduce yank. A write barrier is used during collections to support concurrent marking. 3.7 shows the typical code generated in Go for writes into the heap. Checking whether a collection is currently running requires reading a global variable. During collections the more expensive write barrier is required, otherwise a simple memory store is sufficient.

Algorithm 3.7: heap store

```
1 if collecting then
2 |   writeBarrier(object.field, value);
3 else
4 |   object.field = value;
5 end
```

Compared to other systems Go does not scan the stacks of all threads respectively goroutines during a stop-the-world pause. During the pause Go only sets up the root scanning and marking jobs. Scanning stacks in Go might be quite expensive since there could be hundreds of thousands goroutines and stacks in Go can be arbitrarily large. Go uses a Snapshot-at-the-beginning (SATB) approach with its write barrier and therefore

only needs to scan stacks once. However as described in section 2.4, SATB requires a black mutator.

Go's write barrier solves this by combining the Yuasa [Yua90] and Dijkstra barrier [DLM+78]. As can be seen in listing 3.8, Go marks both the overwritten value and the new referent when writing into the heap. Line 1 implements the Yuasa barrier that allows to use SATB by having black stacks. By marking the overwritten value the mutator can not hide the only reference to an object by moving it from the heap to the stack.

Combining this barrier with the Dijkstra barrier on line 2 is needed since the goroutine's stack is not immediately blackend during the GC pause. Before a goroutine's stack is blackened, the mutator could move the only reference of an still unmarked object from the stack into a black heap object. This would hide the reference from the marking job, Go therefore uses a coarsened Dijkstra barrier to rule out such situations. If the stack would already be black in such situations this would be impossible since all references on the stack would already be marked.

Algorithm 3.8: write barrier

```
1 mark(object.field);
2 mark(value);
3 object.field = value;
```

For improving performance of the write barrier, Go always writes both $object.field$ and $value$ into a local buffer. Only later when the buffer becomes eventually full it calls into the runtime to mark those references reachable. Another consequence of this barrier is that it requires black allocation, hence Go's allocation function marks all allocated objects while a collection is running.

CHAPTER 4

Implementation

This chapter discusses Dora's generational GC implementation and also explains other aspects of the runtime that are relevant for this thesis. Dora is a runtime that parses, type checks and executes source code at runtime. Its programming language is statically typed and makes use of tracing garbage collection for automatic memory management. For good performance Dora relies on compilation to machine code using its JIT-compiler. The compiler is method-based, which means it compiles code on function-granularity. Functions are compiled lazily on the first execution.

Dora's GC is precise, the collector always knows what values in the root set and the heap are actual pointers. In order to find all pointers in the root set, the compiler emits stack maps that determine what stack entries and registers contain valid pointers. This is needed because pointers and values in Dora are untagged. All heap objects start with a type pointer that allows to lookup an object's size and reference fields.

Dora's GC is called *Swiper* and based on the weak generational hypothesis with frequent *minor* collections in the *young* generation and more rare *full* collections for the whole heap. *Minor* collections make use of copy collection, while the *full* collection implements *mark-compact*. *Swiper* aims to be a general-purpose collector by not sacrificing one of throughput, latency or memory usage too much.

While this chapter and thesis is mostly about *Swiper*, Dora has actually more collectors:

1. The **zero** collector never collects garbage but allows for fast bump-pointer allocation in the heap.

2. A pure **copy** collector, which treats the whole heap as a semi space.

3. A *mark*-**compact** collector that compacts the whole heap on every collection.

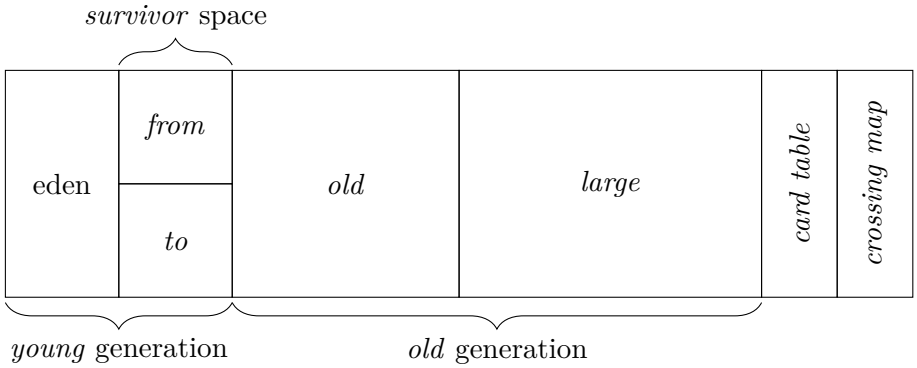

Figure 4.1: Swiper's Heap Layout

These collectors serve as a reference implementation of the respective collection schemes in Dora and are quite valuable for comparisons with *Swiper*. However they are also useful for testing and debugging.

4.1 Heap Layout

When initializing the runtime, *Swiper* reserves contiguous memory for constructing the heap with a given maximum size. Figure 4.1 illustrates the heap partitioning employed by *Swiper*. Since *Swiper* is a generational GC the heap is split into two generations: *young* and *old* generation. However *Swiper* also uses an additional *large* space for objects of size 16K or larger. *Large* objects are never relocated since this usually does not pay off due to the cost of copying. Conceptually the *large* space is considered part of the *old* generation as well, even though those objects are technically in a separate space.

There are even two additional spaces missing in this figure: *code* and *perm* space. These spaces are managed by the runtime instead of each individual GC. The *code* space is exclusively used for allocating executable memory for machine code. Immutable objects are allocated in the *perm* (short for permanent) space. These objects are never collected by the GC.

In addition *Swiper* needs to allocate memory for the *card table* and *crossing map*. The *card table* is about $\frac{1}{512}$ of the heap size, while the *crossing map* is $\frac{1}{512}$ of the *old* generation. The *card table* is used to keep track of *old*-to-*young* references in the collector. The *crossing map* is used for iterating objects starting from a specific card in the old generation. The usage of these two memory areas will be discussed in more detail in section 4.3.

The young generation is further split into two spaces: *eden* and *survivor* space. The *survivor* space is again partitioned into two equally-sized semi spaces.

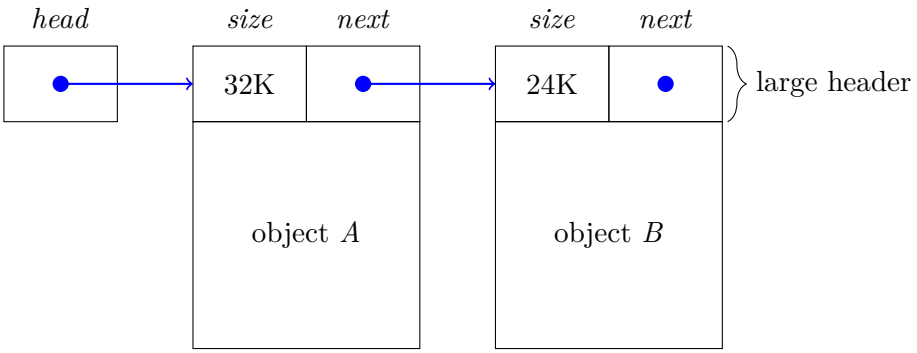

Figure 4.2: Large Object Space

4.1.1 Large Object Space

Large objects are not necessarily allocated in consecutive locations in the large object space. Therefore this space is organized unlike all other spaces. All large objects are tracked in a linked list, the large object space stores the head of the list. Large objects use an extended header that stores pointers to the next object in the list. Figure 4.2 shows such a space with two large objects. In addition to the *next* pointer a header also stores the committed memory size of the current object. This allows cheap access to a large object's size in memory without following the object's class pointer.

The large object space tracks which memory regions in its reserved memory area are currently unused. When a large object is allocated, the collector uses first-fit to find the first free memory region large enough to accomodate the object. The memory area needed for the object is committed. The remaining free memory of this region is still free and can be used for subsequent large object allocations. When the *full* collection finds an unreachable large object, its memory area is added again to the memory regions available for allocation. In addition the collector also discards the pages occupied by the object.

4.2 Object Layout

Each object in Dora starts with a *type* or *class* pointer as its first word. This word is required for heap parsability since it is used for determing an object's size and type. The runtime also reserves an additional word in the object header that can be used for storing additional metadata. *Mark-compact* uses this word to store each object's forwarding pointer and marking bit. Arrays are dynamically sized, their size depends on the number of elements stored and the size of the element type. Therefore arrays need to store an additional word *length* for calculating the size of such objects. This means that on a 64-bit system the smallest object is 16 bytes large (respectively two words). The smallest array with zero elements needs 24 bytes storage or three words.

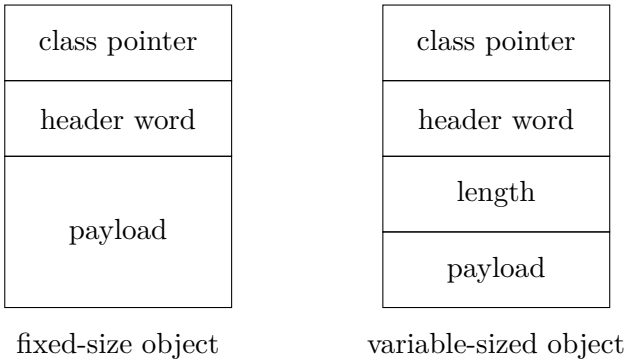

Figure 4.3: Object layout

4.3 Generational Collection

Swiper splits the heap into *young* and *old* generation. Objects are allocated in the *young* generation by default and only promoted into the *old* generation when they survive long enough. This mode of operation is based on the *weak generational hypothesis* - the empirical observation that most objects die young. *Swiper* exploits this property by performing two different kinds of collections. Frequent *minor* collections to reclaim memory exclusively in the *young* generation. But also the less common *full* collection to collect garbage in the whole heap.

Objects of size 16K or larger are allocated in the *large* object space. This space is only collected during *full* collections and therefore considered part of the *old* generation. While smaller objects are always allocated in the *young* generation first, *large* objects directly go into the *old* generation.

4.3.1 Remembered Set

Minor collections in the *young* generation require the GC to track references into this generation. All objects in other generations are retained during a minor collection and therefore might keep additional *young* objects reachable. The data structure that stores all *old*-to-*young* references is the *remembered set*, which is scanned and kept up-to-date during a minor collection. To this end *Swiper* conceptually divides the heap into cards of 512 bytes each. Each such card is either considered *clean* or *dirty*. If a card was marked *dirty*, at least one of the objects stored in the card's respective memory area might contain a reference to the *young* generation. While Dora does not track the precise word storing the *old*-to-*young* reference, it tracks more coarse-grained memory areas that might contain such a reference. If a card is *clean*, there are no references into the *young* generation in the respective memory area.

Section 4.1 already introduced the *card table*, it is used to store each card's state. Since each card is 512 bytes large, the card table is $\frac{1}{512}$ of the heap size.

4.3.2 Write Barrier

The *remembered set* respectively the *card table* in Dora is updated through a generational *write* barrier. The barrier used in Dora was described in [CH93]. When writing a reference into an object, the barrier simply marks the object's corresponding card unconditionally as *dirty*.

Algorithm 4.1: Swiper's write barrier

```
1 object.field = ref;
2 card(object) ← dirty;
```

The compiler emits the write barrier each time a reference is written into an object. Note that the barrier is not needed for non-pointer types. The card's state is indeed updated unconditionally, the barrier does not check whether *object* is actually in the *old* generation and *ref* a young object. As a consequence more cards are marked dirty than required. *dirty* therefore might just mean that some reference was written into a specific memory area since the last collection. Collections in *Swiper* therefore reset a card's state to clean when the card was found to contain no reference into the *young* generation. If this was not the case, *minor* collections might eventually degrade to perform a full scan of the *old* generation.

The card table also needs additional space for the states of cards in the *young* generation, even though the GC never reads their state. Nevertheless the *write* barrier still needs to modify a *young* card's state, this allows the GC to use a cheaper and more coarse-grained *write* barrier.

Another subtle detail of the *write* barrier is that it calculates the card based on the start of an object instead of the modified field's address. With objects spanning multiple cards both approaches might actually update the state of different cards. When scanning a card for references, regular fixed-size objects are always fully scanned as shown in figure 4.4.

This could be problematic with very large objects, typically arrays. In such cases a single write into such an object could force the collector to scan huge amounts of memory. Therefore the *write* barrier is actually implemented differently for arrays compared to regular objects.

Algorithm 4.2: Swiper's write barrier for arrays

```
1 array[ind] = ref;
2 card(array[ind]) ← dirty;
```

The card is calculated from the updated array element instead of the array start. This allows the collector to stop scanning for references at card boundaries for arrays. Figure 4.4 illustrates the difference between regular objects and arrays with respect to the *card*

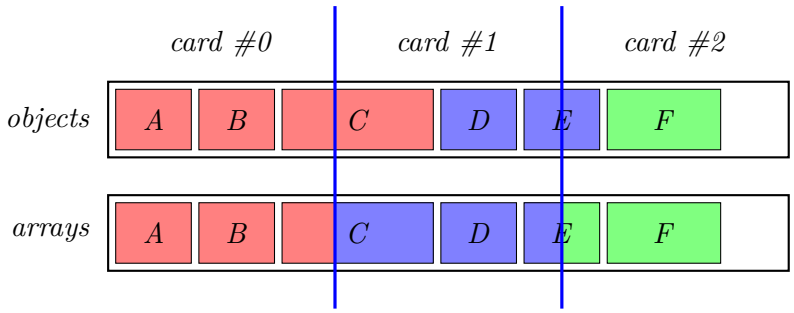

Figure 4.4: Mapping objects and arrays to cards

table. The fixed-size regular objects A, B and C are fully scanned if card #0 is dirty. Even though large parts of C would actually be in card #1. In the case that these objects were arrays, the collector would still fully scan both A and B. However the runtime would only partially scan references in C and stop as soon as card #1 is reached. The part of C that is located in card #1 would only be scanned in the case that card #1 was marked dirty. Basically if the last object in a card is a regular object the full object is scanned. In the case that the last object is an array, scanning of the array stops at the end of the card.

This approach of updating and scanning the *card table* enables the implementation of a very fast *write* barrier. The algorithms 4.1 and 4.2 showed pseudo code for the two kinds of *write* barriers. However the actual implementation was not shown so far, on x86-64 the write barrier can be implemented with two assembly instructions.

```
1 movq %ref, <field offset>(%object)    ; store into field
2 shrq $9, %object                      ; write barrier
3 movb $0, <card table offset>(%object)
```

Listing 4.1: Object write barrier on x86-64

field offset and *card table offset* are constants known to the compiler. *field offset* is the offset of the field in the object, while *card table offset* is the offset added to an object's address to reach its card table entry. In Dora the card table is located right after the heap, therefore the card table entry can be calculated like this:

$$card = card\ table\ address + \frac{object - heap\ start}{512} \tag{4.1}$$

Calculating the card from an object requires the address of the card table and the address of the first object in the heap. Both the card table and the heap start are aligned to the card boundary of 512 bytes. The write barrier avoids the subtraction of the heap start address by using the following two formulas. Only 4.3 has to be calculated in machine code for each individual write barrier in listing 4.1.

$$card\ table\ offset = card\ table\ address - \frac{heap\ start}{512} \tag{4.2}$$

$$card = card\ table\ offset + \frac{object}{512} \tag{4.3}$$

False Sharing Problem

While the *write* barrier is quite fast, it could induce false sharing problems [LTHB14]. False sharing is a problem in multi-threaded applications that can substantially degrade performance. It occurs when multiple threads update distinct memory that is stored on the same cache line. Assuming a cache line size of 64 bytes and Dora's card size of 512 bytes means that the *write* barrier stores to the same cache line for 32KB of contiguous memory. In other words this means that 64 consecutive cards store their state in the same cache line. The compaction of the old generation increases the likelihood of false sharing. This problem not only degrades performance but also the scalability of the application, it becomes worse with the number of threads writing to the same cache line.

It is especially problematic since false sharing on the card table is entirely caused by the implementation. Some developers might not even be aware of this specific implementation detail. What complicates the matter is that the developer has no control over the order of allocation of objects in Dora. An application is even more prone to this problem after a collection compacts the heap. In the case that this becomes a major problem in the future, Dora could switch to another, probably more expensive, *write* barrier. For example the unconditional store to the card could be made conditional.

4.3.3 Crossing Map

When a *minor* collection finds a *dirty* card, objects in this area need to be scanned for references into the *young* generation. Yet objects in the *old* generation are not aligned to card boundaries and might span multiple cards. The *old* generation uses bump-pointer allocation just like the *young* generation, without additional metadata finding a card's first object would require iterating the *old* generation right from the beginning. The *crossing map* provides this information by storing the offset of the first object for each card as a single byte.

While the *card table* needs to reserve an entry for all cards in the heap, the *crossing map* is only needed for the *old* generation and is therefore smaller than the *card table*. This is because *young* cards never need to be scanned. Also the GC knows where *large* objects start and *large* objects are already allocated at card boundaries.

One byte for each card is enough to encode all the information needed about a card. Table 4.1 lists all possible different meanings of a *crossing map* entry. Each time an object is allocated in the old generation that spans multiple cards, the *crossing map* has to be updated. A card consists of 64 words on 64-bit systems, therefore the offset of the first object in the card has to be between 0 and 63. Some cards do not even have

Encoding	Possible values	Meaning
0 - 63	0 - 63	The offset of the first object in the card.
64	1	No references on this card.
65 - 128	1 - 64	Number of leading references in the card before the first object. Used when an object array crosses card boundaries.
129 - 130	1 - 2	An object array starts one or two words before the card start.

Table 4.1: Crossing Map Encodings in Swiper

references at all and therefore marked to store no references. This state is used when allocating regular objects or non-object arrays that span at least three cards. The cards in the middle are marked to have no references. While Dora updates the *crossing map* for these cards in the middle, they should never be marked dirty.

Object arrays are again treated differently to regular objects and non-object arrays. The last card of an object array stores the number of references on this card. The next object in this card starts after the leading references from the object array. When such a card is dirty, the collector first scans the leading references and then continues scanning the subsequent objects. Cards in the middle of the first and last card of an object array are marked to be full with references.

Another complication is caused by the object's layout shown in figure 4.3. An object array could start one or two words before the card boundary. In such cases the first word in the card would either be the array's length or the header word. Both values are certainly not valid heap references and therefore need to be skipped by the collector. *Swiper* therefore uses a different marker when an object array starts one or two words before the card boundary. If the array starts three or more words before that mark, the first word in the card is already a reference again and the *leading references* marker can be used.

4.3.4 Generation Resizing

Having multiple generations in the heap raises the issue of how much space each generation should take. Dora simply reserves the address space for the maximum heap size for each the young and old generation to have maximum flexibility when resizing. The *large* object space reserves twice the maximum heap size to counter possible fragmentation. While Dora reserves address space for a multiple of the maximum heap size, that memory is only committed when needed.

Resizing the young generation might affect performance of an application drastically.

In the case that the young generation is too small, the application suffers from too many minor collections. New objects do not have enough time to die before a minor collection, which leads to many short-lived objects promoted into the mature space. As a consequence the frequency of expensive full collections increases, which again decreases maximum mutator utilization further.

On the other hand increasing the size of the young generation also increases pause times significantly for collections with a high survivor rate. Reducing young generation size can therefore be used to control the average minor pause time of an application. Another aspect is that minor collections are not guaranteed to succeed in *Swiper*. As can be seen in the heap layout in 4.1, a single survivor semi space is not large enough to accomodate all objects from eden and the second semi space. Usually this is not a problem because enough of these objects do not survive a minor collection. Also in the unlikely case that there are too many survivors, objects are promoted directly into the old generation. However even the promotion of objects might fail when there is not enough space left in the old generation. While *Swiper*'s minor collection can handle such situations this is certainly not ideal. This works by keeping young objects in their current location when promotion fails and forcing a full collection immediately afterwards.

Swiper assigns half of the available memory to the young generation after any collection. This ensures that there is enough copy space available such that minor collections succeed. Initially the heap is empty and therefore the young generation takes up half of the maximum heap size. The more objects are promoted the smaller the young generation becomes. A full collection is performed as soon as it becomes smaller than a certain treshold. It is quite important to enforce the lower bound on the young generation size. Otherwise performance suffers through constant minor collections of the small young generation. It might take some time until the heap becomes eventually full with a low survivor rate.

4.3.5 Heap Verifier

An essential part in the implementation of *Swiper* was the heap verifier. The heap verifier is enabled for testing and verifies the correctness of the heap before and after each collection. As can be seen from this whole chapter there are more than enough invariants to satisfy that there is quite some potential for subtle bugs. While strictly speaking the verifier is not needed for functionality, it made the development much easier. Bugs in the collector might be hard to discover and only be observable long after the error happened. The verifier scans each object in the heap and makes sure that various conditions are fulfilled. All outgoing references in objects are checked whether they actually point into the active part of the heap. It also makes sure that references point to the start of another object. In the old generation both the card table and crossing map are controlled to be in sync with the state of the heap. It even verifies that the right memory areas in the old generation are committed and have the proper access rights.

Along with the verifier adding assertions to the collector proved to be quite valuable as

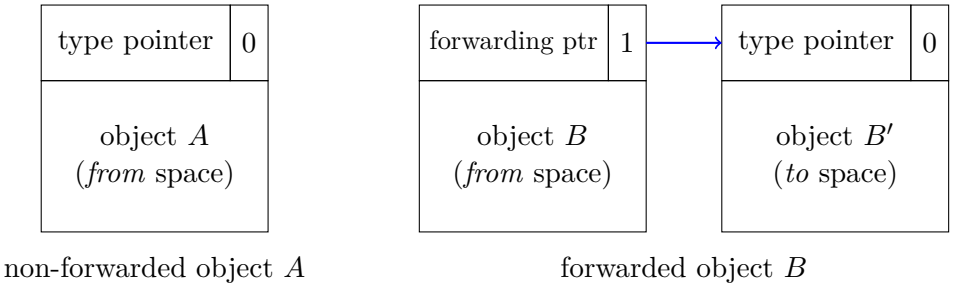

Figure 4.5: Storing the Forwarding Pointer

well. Assertions are used to guarantee assumptions at certain points in the collection. Cheap assertions were added to the release build, while more expensive ones are only active in debug builds of Dora. While running debug builds with the heap verifier enabled takes a lot of time, such tests can easily be run on a Continuous Integration system. This ensures that each commit has to pass all GC-specific tests.

4.4 Serial Minor Collection

Minor collection is implemented using copy collection with the Cheney algorithm [Che70]. This collection copies surviving objects from *eden* and the *from* space to the *to* space. Objects that are old enough are promoted into the mature space.

The mutator allocates objects both in the *eden* and *to* space. At the start of a minor collection *from* and *to* semi spaces switch roles though. The *from* space now contains all objects, while the *to* space is empty. When copying an object into the *to* space, the original object gets tagged as forwarded. The type pointer in each object's first word is replaced with a *forwarding pointer* to the new location of the object. Note that the type pointer information is not lost in the original object since it can always be read from the copy. The collector distinguishes type and forwarding pointer through the value of the least significant bit. Both pointers are guaranteed to be aligned to at least 8 bytes and therefore this bit always contains zero. *Swiper* therefore uses this bit to store whether an object was forwarded already. 0 means that the object was not forwarded and that word still contains the type pointer, while 1 indicates a forwarding pointer (see figure 4.5).

Collection starts by scanning the root set, referenced young objects are copied into the *to* space and marked as forwarded. The object's new location is written into the entry of the root set as well. In case that a young object was already forwarded, only the object's address in the root set entry is updated. References into the old generation are simply ignored.

After scanning the root set, the heap has to be scanned for *old*-to-*young* references as well. This is because *Swiper* is a generational collector and old objects also keep young objects alive. This means that the old generation's card table is scanned for dirty cards. When a dirty card was found, the objects in the card are scanned according to the rules

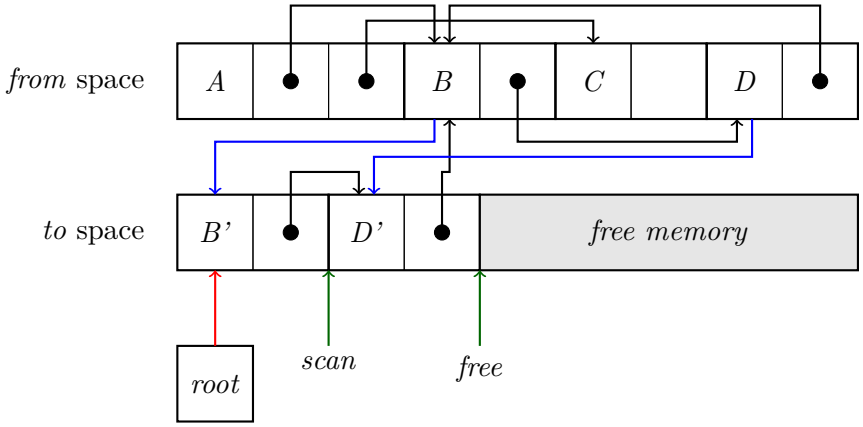

Figure 4.6: Copy Collection with the Cheney Algorithm

explained in 4.3. Again young references are updated through the same logic as used for the root set.

After scanning root set and dirty cards, the minor collection is fully set up. The Cheney algorithm now makes sure that the transitive closure of all reachable objects is copied. The algorithm just requires two pointers: *scan* and *free*. In the beginning *scan* points to the start of *to* space, while *free* points to the first free memory in it. When an object is copied into *to* space, it is copied to the location referenced by *free*. *free* is then advanced by the copied object's size to reference the first free memory address again.

scan conceptually divides the *to* space into two areas. Objects located before *scan* are black objects, all these objects were already scanned for young references. That means all young references already point into the *to* space. All objects at *scan* or afterwards are gray objects. These objects were already identified as reachable and therefore copied into *to* space. However outgoing references in them still have to be scanned for further young references. Allocating objects moves *free* forward, while scanning objects for young references moves *scan* forward. The copy collection finishes when *scan* and *free* meet. Then no more objects are left that need to be scanned.

Figure 4.6 showcases this algorithm on a simple heap. The root set only referenced object B, B was therefore copied into the *to* space as B'. After copying B, the root set now references B'. Next object B' was scanned for more young references, which in turn copied D into the *to* space as D'. D' is the next object to be scanned, it still references B in the *from* space. After updating this reference *scan* and *free* meet: the collection is finished.

Not all young objects marked reachable are copied into *to* space however. Objects old enough are promoted into the old generation. Object age is based on the number of minor collections each object has survived. As soon as an object survives the second minor collection it is promoted in *Swiper*. Unlike other collectors, Dora does not store the age for each object individually. The field *age marker* is updated after each minor

collection to the address of the free memory in the *to* space. This is similar to how V8 operates. Objects before that marker have already survived one minor collection and are going to be promoted in the next cycle. While objects afterwards were just recently allocated and stay in the young generation.

Note that promoted objects need to be scanned as well, therefore both the young and the old generation have a *scan/free* pointer-pair. In fact *Swiper* has its own pair of pointers for each region in the old generation. As will be explained in section 4.8, parallel full collection splits the old generation into multiple regions. Scanning of old objects is more involved than for young objects: The collector also needs to test whether a promoted object contains pointers into the young generation after scanning. In this case the card has to be marked as dirty. As previously mentioned in section 4.3 regular objects and arrays follow different rules here.

4.5 Serial Full Collection

Full collection in *Swiper* uses the *mark-compact* collection scheme. Live objects are moved towards the start of the old generation. Compaction uses the Lisp 2 algorithm described in 2.1.3. Collection is performed in 4 distinct phases: 1. marking, 2. compute forwarding pointer, 3. update references and 4. relocate. Since the general algorithm was already discussed, this section focuses more on the specific implementation details in *Swiper*.

The *marking* phase marks all reachable objects in the heap, hence young, old and large objects. In contrast to the minor collection, the full collection collects garbage in the whole heap. The bit required to store whether an object was found reachable is stored in the least significant bit of the header word in each object.

In the *compute forwarding pointer* phase the collector calculates the new location for young and old objects. Similar to the marking bit, the forwarding pointer is also stored in the header word of each object. This is possible because the forwarding pointer is aligned and therefore the least significant bit is always zero. First the collector arranges live old objects towards the beginning of the heap. All live young objects are promoted into the old generation. This means that both the eden and survivor spaces are completely empty after a full collection. The surviving young objects are located immediately after the reachable old objects. The order is important here since otherwise young objects would overwrite still required objects in the *relocation* phase. Large objects are not relocated and therefore not traversed in this phase.

Swiper could consider the age of young objects similar to the minor collection. This would allow the full collection to only promote some objects based on their age, while objects allocated after the last collection would be copied into the survivor space. However the problem is that this would complicate the implementation significantly. Assuming that a full collection promotes all young objects, it is enough to clear the full card table. As soon as objects remain in the young generation this is not possible anymore. The

implementation would have to check whether objects contain young references and change the card state accordingly.

The next phase then scans the heap again to *update* all outgoing *references* in all live objects. References need to be updated for all generations, respectively spaces: young, old and large objects. All references in the root set are forwarded during this phase as well. When traversing large objects, dead large objects are removed from the space and their memory is freed as well. This avoids traversing large objects twice, however dead young and old objects are simply skipped. At the same time cards are cleared for both live and dead large objects.

In the final phase objects are relocated to their destined location. In addition this phase also updates the crossing map for the old generation. At the very end the card table is reset to clean for the old generation.

4.6 Parallel Marking

Marking is one of *Swiper*'s most expensive full collection phases and therefore predestined for further speedup through parallelization. Algorithm 4.3 illustrates marking implemented as a serial algorithm. While *Swiper* follows the tri-color abstraction, it does not explicitly store the color gray for each object. As previously mentioned Dora only stores a single marking bit in the header word for each object. This bit simply determines whether an object was marked reachable or not. The collector would have to inspect whether the worklist contains the object to decide whether an object is actually gray or black. This would be quite expensive, however *Swiper* right now does not need to perform such tests at the moment. If the collector would need these in the future, it could simply use a second bit in the header word for storing whether an object is *gray*.

Algorithm 4.3: Serial Marking

```
1  worklist ← ∅;
2  for root in rootSet do
3  |   mark(root);
4  |   worklist.push(root);
5  end
6  while worklist non empty do
7  |   object ← worklist.pop();
8  |   for field in fields(object) do
9  |   |   if field not marked then
10 |   |   |   mark(field);
11 |   |   |   worklist.push(field);
12 |   |   end
13 |   end
14 end
```

Performing marking on multiple threads is not straightforward since work can not be partitioned statically. The load between multiple worker threads has to be balanced dynamically. To this end parallel marking is implemented using a work-stealing technique such that work is shared more equally between workers.

The basic setup of the parallel algorithm works as follows: *Swiper* creates a pool of worker threads that perform all the necessary work. Marking is finished when all worker threads run out of work and quit. Each worker thread has to acquire units of work and run them. In this parallel marking algorithm the unit of work is a single object address. A worker thread scans all outgoing references of that object and marks them reachable. Objects that have just been found to be reachable are pushed again onto the worklist such that all transitive reachable objects are eventually marked. Work can come from multiple sources in this implementation:

1. Each worker has a work queue that it uses to push and pop work from. Other worker threads can only *steal* work from this queue.
2. If the local queue is empty, the worker tries to retrieve work from the global work queue. The global work queue is shared by all threads. Initially this queue is filled with the objects referenced from the root set.
3. As a last resort, the runtime tries to steal work from other threads. Stealing works by trying to acquire work from another thread's local worker queue. The collector randomly chooses a thread to steal from [Hel12]. Stealing is repeated until it eventually succeeds or hits an upper limit of attempts. Each worker tries to steal work $2n$ times, where n is the number of worker threads.

If the thread could not acquire work from any of these sources, it tries to quit. The algorithm used for suspending workers is described in 4.6.2.

4.6.1 Chase-Lev Deque

The worklist used by each worker thread is a lock-free Chase-Lev work-stealing deque [CL05]. *Swiper* uses the implementation of the deque from the Rust crate *crossbeam*[1]. Utilizing this widely-used library simplified implementation a lot; it is notorioulsy hard to ensure the correctness of lock-free data structures. Especially when the support of multiple architectures with different memory models is needed, as in the case of Dora [LPCZN13].

The Chase-Lev deque is a double-ended queue, it allows cheap insertion and removal at one side and stealing from the other end. For this end it provides three different operations: *pushBottom*, *popBottom* and *steal*. Only the worker thread itself is allowed to use *pushBottom* and *popBottom* on its deque. While *steal* can be used by all other

[1] `https://github.com/crossbeam-rs/crossbeam`

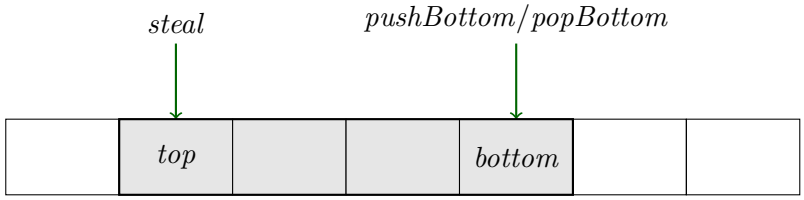

Figure 4.7: Chase-Lev Deque

threads to remove elements from the top of the deque. Stealing from such a deque requires synchronization. On the other hand elements can be inserted and removed at the bottom of the deque without synchronization unless the last element gets removed. Not requiring synchronization for the worker in most cases makes these operations quite efficient. This data structure is backed by a circular array that is allowed to be grown. When an element cannot be inserted into that array anymore, *pushBottom* allocates a fresh and larger array. All data is transferred to the new array, subsequent operations are performed on this array.

Even though updating the deque from the worker is cheap, each worker has an additional local worklist as well. This has proven to improve pause times during evaluation. That local worklist is a simple vector, objects are primarily pushed and popped from this stack without any synchronization at all. Only when an array would not fit into the worklist anymore, the work-stealing deque is used. Entries in the local worklist cannot be stolen from, therefore a worker thread pushes half of its local worklist into the global worklist after some time. This allows other threads to assist in marking theses objects reachable.

4.6.2 Termination Protocol

In this parallel algorithm, threads cannot simply quit when no more work could be found. It could happen that there is still some thread with local work that cannot be stolen, while all other threads run out of work. That thread could discover a huge new subgraph that needs to be marked. In this case this thread then would have to mark all objects itself.

Therefore *Swiper* uses a termination protocol such that threads do not quit until all threads have run out of work [FDSZ01]. The runtime uses a counter that is initialized to the number of threads. When a worker thread fails to acquire more work, it atomically decrements that counter by 1. Afterwards the thread now waits a short amount of time to check the counter again and see if all other threads are ready to quit as well. If the counter is zero after this timeout, all other threads have run out of work as well, therefore all threads are allowed to quit. If this is not the case and the counter is still non-zero, the worker atomically increases that counter again by 1. Termination is cancelled and the thread repeats the whole process of acquiring work again.

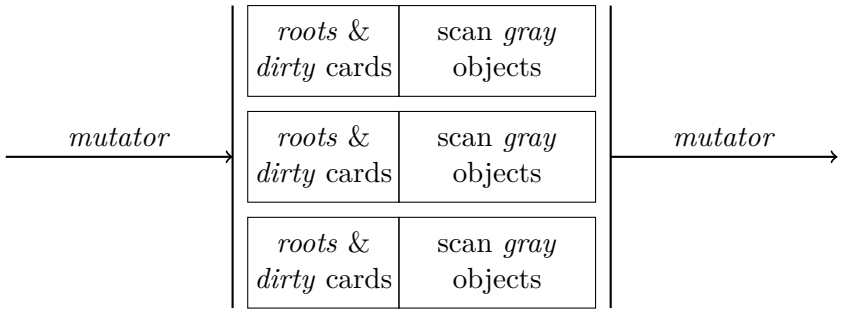

Figure 4.8: Parallel Minor Collection

4.7 Parallel Minor Collection

Swiper also has a parallel version of the single-threaded minor collection [FDSZ01]. Both approaches are completely separate implementations: while there are certainly many similarities between them, they are different enough to warrant two distinct versions. This ensures that neither of them holds back the other or performance suffers for one of them.

In the parallel minor collection *Swiper* performs all the work during a single pause again. However, this time the collector uses multiple threads for distributing work. The goal of this approach is to reduce pause times for reclaiming memory in the young generation. The approach uses dynamic load balancing, utilizing the same technique already known from parallel marking (see 4.6). For parallel marking the worker thread marked all outgoing references of an object and pushed them again onto the worklist. In the minor collection, workers scan an object for outgoing references into the young generation. They either copy young objects into *to* space or promote them and then append them to the worklist. In addition they also update references to the young generation to the new location of the object. This already clarifies that both serial and parallel minor collection follow the same architecture. For example the same objects are promoted, albeit in a different order.

The runtime starts the worker threads immediately after pausing the application. The threads perform two separate phases: 1. scanning of roots & dirty cards and 2. scanning of gray objects (see figure 4.8). Processing of the second phase cannot start before all threads have not finished phase one. These two phases are separated using a barrier. Only after every worker thread has reached that barrier, execution continues. The reason for this was to not have cleaning of cards in the first phase and dirtying of cards in the next phase interfere with each other. Note that while roots are evacuated in parallel, determining the root set is still single-threaded in the runtime.

Scanning the old generation for dirty cards is parallelized as well, since this could become quite expensive for large old generations. Cards are therefore divided into N strides: the first stride would be the cards $\{0, N, 2N, \ldots\}$, the second stride $\{1, N+1, 2N+1, \ldots\}$. N in this context is a multiple of the number of worker threads used, cards are therefore

overpartitioned. The reason for this approach compared to dividing all cards into contiguous ranges is that according to [FDSZ01] dirty cards tend to occur in clumps. This approach therefore divides work more equally in such cases.

However cards for large objects are scanned at once by the thread that obtained that object. Traversing all large objects is distributed between worker threads using a simple Mutex. The first thread to lock the mutex acquires the next large object. In the future it might become apparent that this design choice has too much synchronization overhead. In this case one thread could traverse all large objects without synchronization and then distribute work to the other threads in similar sized chunks. Alternatively strides could be used for large objects as well. Both approaches would allow distributing scanning of very large objects.

4.7.1 Forwarding Pointer

Updating the forwarding pointer stills works as illustrated in figure 4.5. This time the collector needs to take special care to update the pointer atomically. The worker thread first allocates a new location for an object and then copies the object to it. Afterwards it tries to atomically update the forwarding pointer in the original object. For this an atomic compare-and-swap operation is used: the operation is only successfull if the first word in the object still stores the object's type pointer. If this fails, another thread has already replaced this word with another forwarding pointer. Since the other thread was first with installing the forwarding pointer, the allocation can be undone if possible. The object's new location was therefore defined by another thread. This approach ensures that each young object only has one active copy.

4.7.2 Local Allocation Buffer

All workers perform sequential allocation both in the young and old generation. This requires synchronization between all threads for the allocation of each object. This would be quite expensive, therefore each worker thread allocates so-called local allocation buffers. Subsequent allocations can be served from within this buffer without requiring synchronization. This is only needed for the allocation of the buffers itself. Each thread needs distinct local allocation buffers (LAB) for both the young and old generation. However, the runtime only needs to obtain a buffer when the first object in this area needs to be allocated. This is illustrated by figure 4.9, where the third worker has not allocated a LAB for the old generation yet.

As soon as the first object does not fit into a LAB anymore, a fresh one needs to be allocated. It is not unlikely that the old LAB still has unallocated memory at the end of the buffer. *Swiper* needs the heap to be parsable, so the worker fills the free memory with an unused filler object. The object is then allocated in the new LAB.

LABs introduce a certain amount of fragmentation in the heap. The collector tries to reduce this by only allocating small objects in the LAB. Therefore the gap at the end of a LAB should not be too excessive. In the contrast to small objects, medium-sized

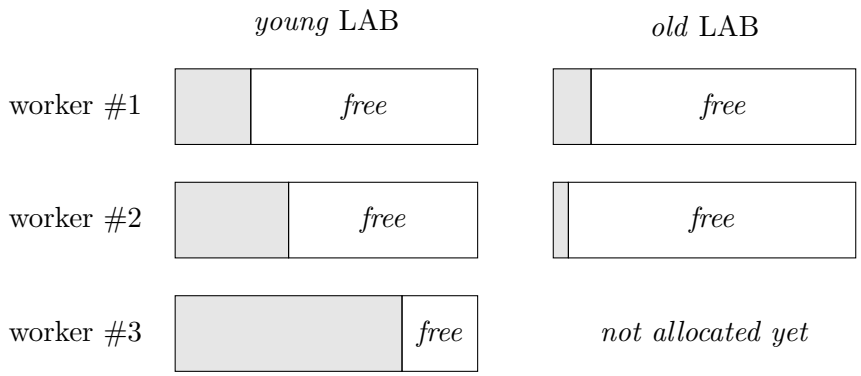

Figure 4.9: Local Allocation Buffers

objects are allocated outside of LABs. Medium-sized objects are small enough to not be allocated in the large object space but they are still assumed to be less frequent than small objects. This should mean that still requiring synchronization for them is acceptable for performance.

4.8 Parallel Full Collection

Swiper also reduces pause times for full collections by distributing work to multiple threads. Similar to the minor collection Dora supports serial and parallel full collection through two separate implementations. *Mark-compact* is still composed of the four phases known from the serial version: 1. marking, 2. compute forwarding pointer, 3. update references and 4. relocate. The marking phase is parallelized using dynamic load balancing and work-stealing as previously described in 4.6.

4.8.1 Old Generation Layout

The major difficulty of this approach is ensuring that still required objects are not overwritten by another thread in the parallel relocation phase. To this end the old generation is split into multiple regions. Instead of relocating objects to the start of the heap, each object is only relocated within its surrounding region. Each worker thread takes over a region and relocates all live objects within it. This ensures that copying objects does not conflict with other worker threads.

Each region starts with an allocated memory area, optionally followed by free memory. As can be seen in figure 4.10, the regions cover all of the old generation's memory. Initially the old generation only consists of one region, the first parallel full collection will then divide it into multiple regions. Scanning all objects in the old generation requires scanning from the start of each region. Regions are object-aligned but presumably not card- or page-aligned.

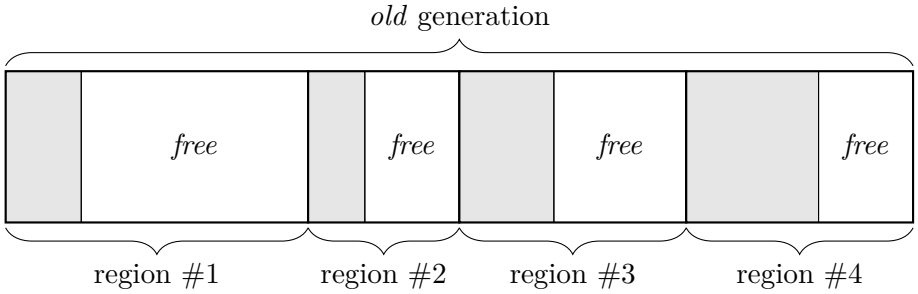

Figure 4.10: Regions in Old Generation

4.8.2 Compute Forwarding Pointer

All phases in the parallel full collection use a pool of worker threads for distributing work. To make this possible the heap is separated into about $8n$ approximately equal-sized units, where n stands for the number of workers.

Units are memory regions completely filled with either live or dead objects. The collector guarantees that units are object-aligned, this means each object in the young and old generation belongs to exactly one unit. Creating old generation units requires access to the crossing map for finding object boundaries in a card. The young generation is split into exactly three units at the moment: an unit for the *eden*, *from* and *to* space. This is because there are no crossing map entries for the young generation and therefore splitting those spaces into multiple units would require linear scanning of each space at the moment.

As soon as the collector has appointed all units in the heap, the worker threads scan each unit for the exact number of live bytes in it. The worker thread simply scans objects in the unit and increases the number of live bytes if an object was marked reachable in the previous *marking* phase. This is the only additonal work needed in the parallel collection but not in the single-threaded implementation. Based on the number of live bytes in each unit, units in the old generation are partitioned into n regions of approximately equal size. Each region consists of one or more consecutive units and is therefore object-aligned as well. Note that while units in a region need to be consecutive, there might still be gaps of free memory between them. As a result regions before and after a full collection do not necessarily match, they might grow or shrink in their maximum boundaries. This is illustrated in figure 4.11, where the new second region consists of units of region #1 and #2 before the collection.

A region spans the memory area from the start of the first unit to the end of its last unit. It is paramount that each region only moves objects strictly within these bounds, otherwise data from other regions is overwritten. After computing regions from the old generation units, the young generation units are placed in one of the regions. These young units are evacuated and therefore the collector has more freedom when relocating them. The collector simply assigns a young unit to the first region that is able to accomodate

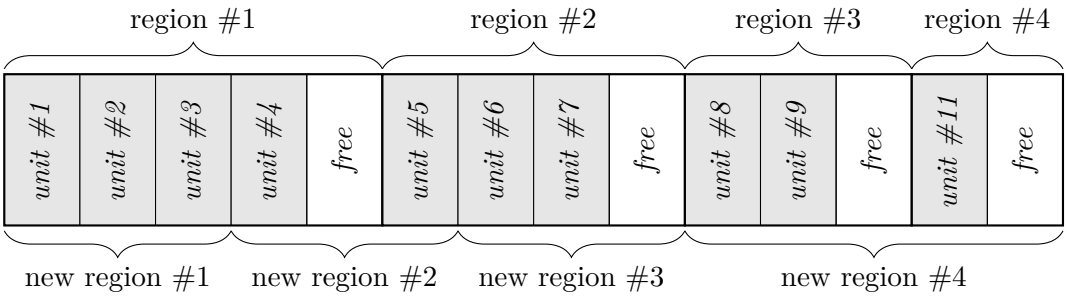

Figure 4.11: Assigning Units to Regions in Old Generation

its live objects. The surviving young objects are located right after the compacted old objects of a region.

It is also important to point out that regions are only guaranteed to be object-aligned but not card-aligned. This means regions might have live objects on the same card. As a consequence the first and last card of all regions are not cleared during serial and parallel minor collections. Marking the last card of a region clean, could also mistakenly clean the next region's first card. The collector would then miss references into the young generation and wrongly reclaim memory.

Now that both units and regions are determined, the collector can finally calculate the new locations of live objects in the heap. Objects in a region are slided towards the start or end of the region. Regions are relocated in alternating directions to have larger blocks of contiguous free memory available after the collection. Figure 4.12 shows that this makes it possible to reduce the number of regions after a collection. Regions during and after the collection are actually different in *Swiper*. Workers obtain an unprocessed unit and assign new locations to live objects. This is feasible without synchronization since both the new start address of the unit and the total number of live bytes in it is known to the runtime.

4.8.3 Update References

The next phase updates all outgoing references in live objects by following the forwarding pointer. This phase is quite similar to the serial version in 4.5. The full task is again distributed as units to the individual worker threads. Apart from obtaining units no synchronization is needed for young and old units.

After processing units large objects are iterated by all worker threads using a simple Mutex, this is quite similar to the mechanism described for the parallel minor collection in 4.7. The collector traverses large objects only once in collections and therefore performs more work for them. Initially the card table entries for all large objects are reset to clean. Dead large objects are immediately freed during the traversal, while in reachable objects references are updated. Live objects are appended to a local list of large objects for each worker thread. As soon as a worker thread is finished, it appends its local list to the

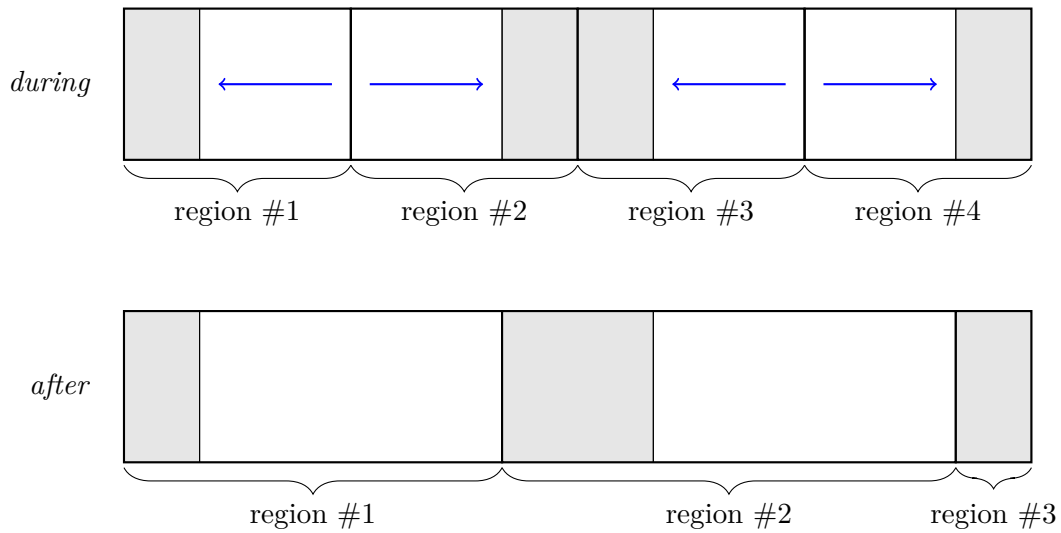

Figure 4.12: Sliding Regions in Alternating Directions

global list used for the large object space. Synchronization is therefore only needed once at the very end for each worker thread to construct a single list of all live large objects.

4.8.4 Relocation

The final phase relocates objects according to each object's forwarding pointer. This time worker threads acquire regions as the unit of work to guarantee that units are evacuated in the right order. Regions that are compacted to the start, need to relocate the left-most unit first. When compacting regions towards the end, the right-most unit is processed first instead. This order assures that no required data is overwritten. Since units for the young generation are completely evacuated, these units are copied after all old units in this region were processed.

In addition to relocating objects, this phase also cleans cards for the old generation. Work is divided based on regions, with the boundaries that were used before the current full collection. As soon as worker threads have relocated objects, they try to acquire regions to clean their cards. As previously mentioned, the cards for large objects were already cleaned when updating references.

4.9 Thread-local Allocation

Another important aspect of a collector is the allocation of objects. *Swiper* supports fast bump pointer allocation both in the young and old generation. However this still requires the mutator to call into the runtime on every allocation. In addition multiple threads might be allocating at the same time and therefore object allocation needs to be synchronized using an atomic Compare-and-Swap operation. Although Dora is

single-threaded right now, allocation in Dora is already synchronized to allow for simpler implementation of multi-threading features in the future.

Hence Dora provides thread-local allocation to improve allocation throughput. Instead of allocating individual objects each thread allocates an allocation buffer from the runtime. When the mutator then tries to allocate an object, the request is served by allocating memory from within the buffer. No call into the runtime or synchronization is required anymore. Only when the buffer becomes full, the application calls into the runtime again to acquire a new buffer. With a thread-local allocation buffer (TLAB) allocating a new object becomes extremely cheap.

4.9.1 Implementation in the Runtime

Thread-local allocation needs to be supported both by the collector and the runtime. Dora needs two additional pointers for implementing a TLAB: `tlab_top` and `tlab_end`. `tlab_top` stores the address of the next allocated object and is incremented at each allocation by the object's size. The second pointer is `tlab_end` which stores the end of the current allocation buffer. Allocation succeeds as long as `tlab_top` is less than or equal to `tlab_top`. As soon as allocation fails, another call into the runtime for a new allocation buffer has to be performed. However this is now much more infrequent than without TLABs and helps quite a lot in many benchmarks. To reduce fragmentation, only small objects are allocated in a TLAB. Medium-sized allocations still require the more expensive allocation.

The runtime also needs to make these pointers easily accessible from the compiled code. In Dora each thread has its own instance of the *ThreadLocalData* structure (see listing 4.2) that can be used to store both pointers. Dora keeps a pointer to the current thread's *ThreadLocalData* in a specific register at all times in compiled code. This register needs to be initialized when calling Dora functions from native code, but also needs to be saved before calling into native code and restored after returning from native code into Dora-compiled code again.

```
1 pub struct ThreadLocalData {
2     d2n: *const DoraToNativeInfo,
3     tlab_top: Address,
4     tlab_end: Address,
5 }
```

Listing 4.2: Thread-local Data

Loading `tlab_top` and `tlab_end` is therefore a simple load instruction:

```
1 movq <offset tlab_top/tlab_end>(%thread_register), %dest
```

Listing 4.3: Loading tlab_top and tlap_end

Storing into `tlab_top` is again just a simple store instruction, `tlab_end` does not need to be modified by compiled code but can be initialized directly by the GC.

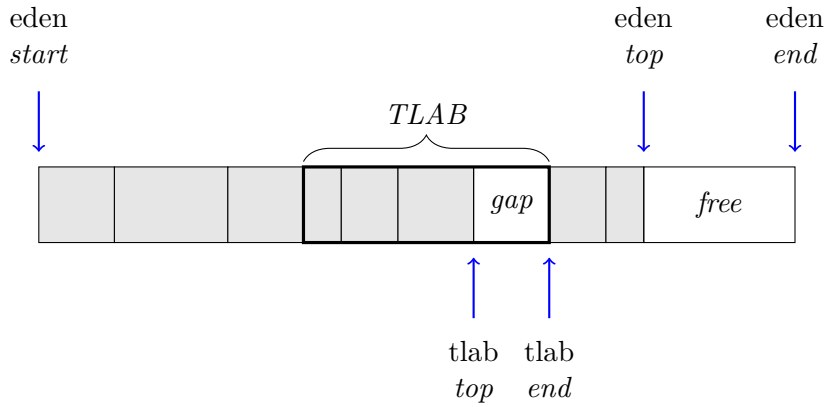

Figure 4.13: Gap in TLAB

4.9.2 Heap Parsability

Heap parsability is an important property since it allows walking through all objects in a heap simply by increasing a pointer by the current object's size. With TLABs the heap is not as easily parsable anymore since there might be gaps at the end of the buffer (see figure 4.13). This happens anytime the buffer is already too small to allocate the next object. In this case the runtime fills the TLAB with an unused object that spans the rest of the buffer up to its limit. This makes the heap parsable again. Depending on the size of the gap a different object type is used.

As previously mentioned the smallest object spans at least two words. Nevertheless there are going to be cases where the gap is only one word wide. In this cases the runtime stores *null* in this class pointer word. When the collector later tries to walk through all objects in the heap, it needs to check for a *null* class pointer and increment the pointer by one word when encountered.

If the gap is two words wide, a regular object is written into this location. Larger gaps of three or more words are filled with integer arrays. Even in the worst case for large integer arrays the runtime has to perform at most three memory writes: the data in an integer array is not cleared. This is not necessary since no other object can reference the array and therefore not read its elements. Also using an integer array means that the object has no outgoing references. Therefore the runtime can simply skip such objects when scanning them for references.

CHAPTER 5

Evaluation

This chapter evaluates *Swiper*'s serial and parallel collections. Various metrics are compared to other basic collection schemes implemented in the Dora runtime.

5.1 Methodology

Various aspects of GC performance are measured in this evaluation. For this end, benchmarks are run 20 times and the average of those runs is reported as result. Pause and execution times are recorded by the Dora runtime itself using the system's monotonic clock.

Benchmarks are run after a reboot with no other applications running. To reduce the number of running processes on the machine, the operating system is booted into console without a GUI. For simplicity simultaneous multithreading (SMT) is disabled for all benchmarks even if they will create more threads than the CPU has cores. Also Turbo Boost is disabled to preclude differences in clock frequency between cores and different benchmarks.

5.1.1 Configuration

Dora was compiled using the latest Rust Nightly version 1.36.0 (from 2019-04-24). Note that the release build of Dora is used. Additionally checks like the heap verifier are disabled throughout this chapter.

5.1.2 Test Environment

All tests were performed on the same system using an AMD Ryzen 2700X with 3.70GHz. The CPU has 8 cores and 16 threads, which come handy when testing *Swiper*'s parallel collections. The system has 32GiB DDR4 main memory with 3200Mhz bus speed. A

Samsung NVMe SSD 970 EVO is used for disk storage. The operating system is Fedora 29 with Linux 5.0.7.

5.2 Benchmark

This section discusses the benchmarks used throughout this evaluation. At the moment there are no large real-world applications written in the Dora programming language. Translating such huge programs to Dora would be infeasible in the time budget for this thesis. Therefore this chapter uses synthetic micro-benchmarks for measuring. Nevertheless these benchmarks have interesting GC behavior and stress both allocation and collection. They were also used for evaluation in other GC-related papers. The benchmarks were translated to Dora from Java, JavaScript and C# programs.

5.2.1 gcbench

This application was translated from Hans Boehm's GC benchmark, it was also used in [ABCS03] for performance evaluation. It creates trees of various sizes both from bottom-up and top-down. Trees also have different lifetimes, initially a large tree is allocated that remains live until the end of the application.

5.2.2 binarytress

binarytrees is an adapted version of *gcbench* and was taken and translated from the *Computer Language Benchmark Game*[1]. The benchmark is also studied in detail in [FCJH16]. Initially this application creates a large short-lived tree to stretch the heap. Afterwards a slightly smaller tree is created that is kept alive during the whole execution.

In each iteration the application then creates a certain number of short-lived trees of the same size. With each iteration the application creates fewer but larger trees. During the creation of very large trees the GC is likely to perform multiple collections. Hence such trees are promoted into the old generation.

5.2.3 gcold

gcold is a benchmark for testing the old generation and major collections. Initially an array of trees is created, where each tree is about one megabyte in size. In the configuration used in this thesis, the array contains about 300M of data. After the initial setup of live data, the benchmark executes a specific number of steps. In each step short-lived data is allocated, that immediately becomes garbage in the next step. In addition random trees and subtress of the long-lived array are replaced with fresh ones. This means part of the old generation becomes garbage and some newly allocated data lives long. For every byte of long-lived data, 3 bytes of short-lived data are allocated.

[1] `https://benchmarksgame-team.pages.debian.net/benchmarksgame/`

In each step the application then also performs mutations on the long-lived data. This benchmark application was used in [FDSZ01] in a similar configuration.

5.2.4 splay

This benchmarks is part of Google's benchmark suite *Octane*[2] and was translated from JavaScript. The application implements a *splay* tree. A *splay* tree is a balanced binary tree that allows cheap access to recently accessed data. The benchmark initially creates a tree of random values with 32000 nodes. Then the application inserts repeatedly new nodes and removes other ones again.

5.2.5 splunc

splunc was used in [SBH05] and implements a *splay* tree as well. This benchmark inserts repeatedly new nodes into the *splay* tree and then truncates the tree at a depth of 50 nodes. Additionally the nodes that are inserted into the tree vary in their size.

5.3 Serial Swiper

In this section the benchmarks *binarytrees*, *gcold* and *splay* are run using different GCs: *Copy*, *Compact* and the generational *Swiper* with its serial collections. *Copy* implements copy collection, whereas mark-compact is used in *Compact*. Each benchmark is executed with multiple heap sizes to expose potential differing GC behavior. Note that when running the benchmark using the *Copy* collector, Dora is executed with twice the heap size. This is certainly unfair to the other collectors, however allows for simpler comparison in this thesis.

For each benchmark the total execution time, collection time and mutator time is recorded. The collection time is the time spent collecting garbage in the GC, hence it is the sum of all GC pauses. Whereas the mutator time is the time spent in the application including allocation. In addition L1 data cache misses and data TLB misses were recorded while the mutator was running. These metrics are useful to explain differences in mutator time.

The results are depicted in the figures 5.1-5.15. As expected *Swiper* spends more time in the mutator compared to *Copy* and *Compact*. This is partly because of the write barrier used in *Swiper* and depending on the benchmark due to cache misses. *Swiper* performs quite well for *binarytrees* and *gcold*, its execution time is comparable to *Copy* while using half the maximum heap size. However for *splay*, *Swiper* is even slower than *Compact*: *splay* violates the *weak generational hypothesis*. For this reason *Swiper* needs to copy data more often than *Compact* to promote objects into the old generation. This demonstrates that *Swiper* works best in generational workloads. Even though performance degrades for *splay* with *Swiper*, the regression is still acceptable.

[2] `https://github.com/chromium/octane`

Tables 5.1 and 5.2 contain more information about the individual GC pauses in *Copy* and *Compact* during the benchmark. The results for both collectors show that increasing the heap size reduces the number of collections. However increasing the heap size also increases the pause time when a collection finally happens. Increasing the heap size only postpones collections, but memory has to be reclaimed eventually. Nevertheless the collection time is still reduced overall when increasing the heap size.

Tables 5.3, 5.4 and 5.5 illustrate the pause times and the duration of individual phases for minor and full collections in *Swiper*. Depending on the benchmark, respectively the amount of live data in the heap, either *marking* or *compute forward* is the most expensive phase of the full collection. According to the results in these benchmarks, the minor collection is indeed much more frequent than the full collection. However, unlike *Copy* and *Compact* the number of collections is not reduced for heap sizes of *500M* and *550M* for *binarytrees*. This spike in minor collections can also be observed in the total execution and collection time for the benchmark (see 5.1 and 5.3). The fact that the average minor collection pause decreases significantly though, helps understanding the reason for the regression. *Swiper* uses a simple heuristic for deciding between minor and full collections: if the young generation is smaller than 1M, a full collection will be performed. In some cases the young generation is only slightly larger than that and minor collections do not promote enough objects such that the threshold is not reached for some time. Hence *Swiper* performs a lot of minor collections with quite small young generations. This behavior can be avoided by increasing the minimal young generation size, this would force *Swiper* to collect the full heap before running into this pathological case. However a better long-term solution would probably be to implement a more sophisticated heuristic for this.

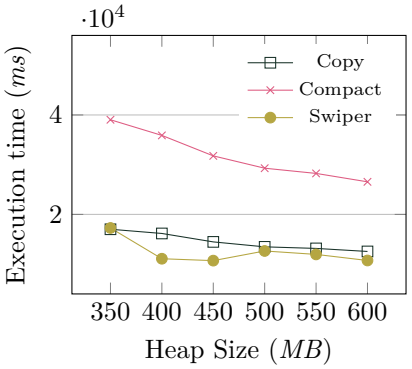

Figure 5.1: *binarytrees* Total Time

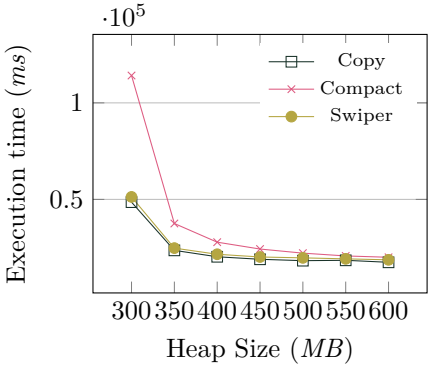

Figure 5.2: *gcold* Total Time

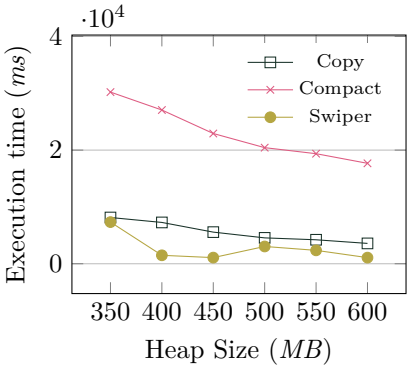

Figure 5.3: *binarytrees* Collection Time

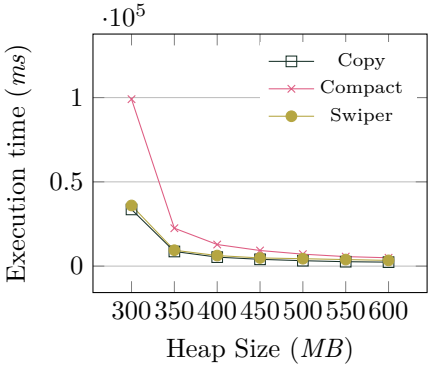

Figure 5.4: *gcold* Collection Time

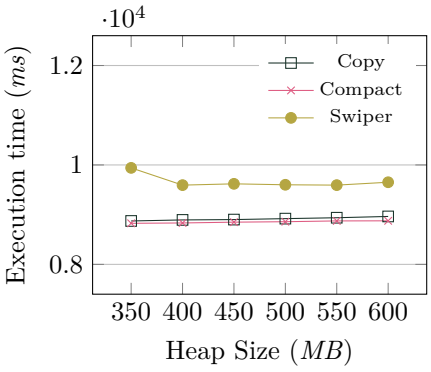

Figure 5.5: *binarytrees* Mutator Time

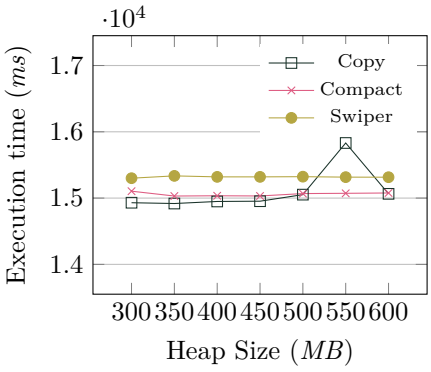

Figure 5.6: *gcold* Mutator Time

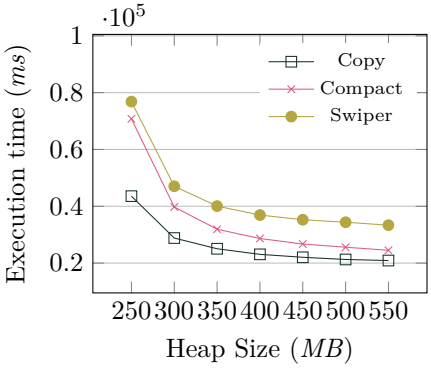

Figure 5.7: *splay* Total Time

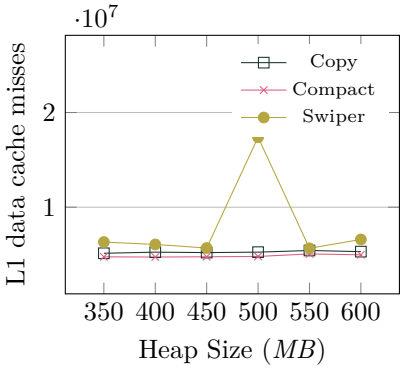

Figure 5.8: *binarytrees* L1 data cache misses

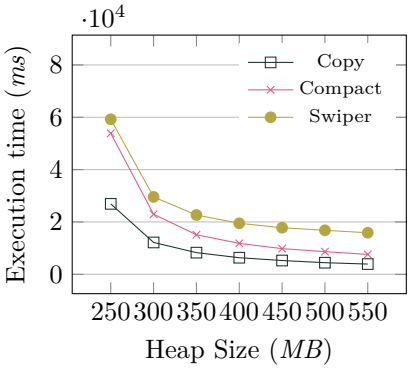

Figure 5.9: *splay* Collection Time

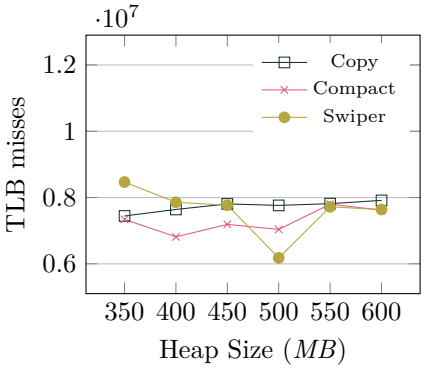

Figure 5.10: *binarytrees* TLB misses

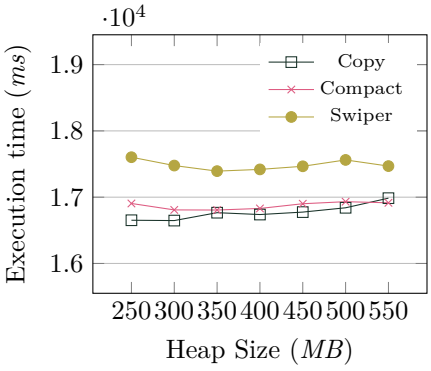

Figure 5.11: *splay* Mutator Time

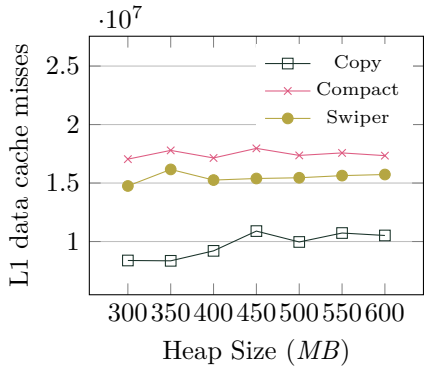

Figure 5.12: *gcold* L1 data cache misses

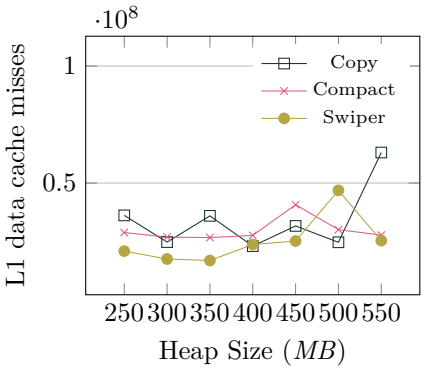

Figure 5.13: *splay* L1 data cache misses

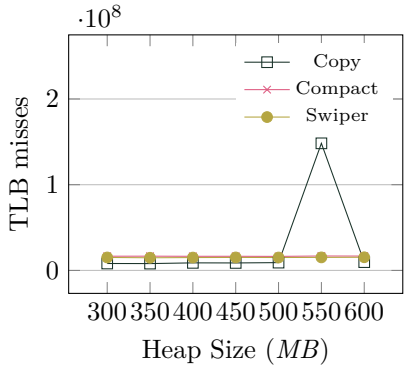

Figure 5.14: *gcold* TLB misses

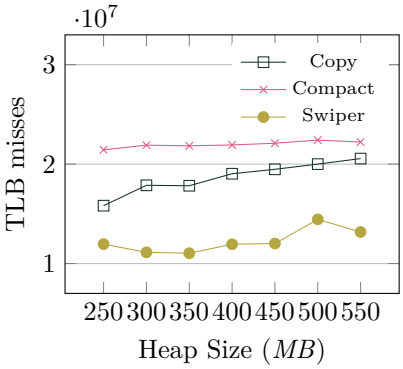

Figure 5.15: *splay* TLB misses

Heap Size	btrees	gcold	splay
250M	⁄	⁄	Count 281 Avg 95.0ms Min 92.1ms Max 196.1ms
300M	⁄	Count 275 Avg 123.0ms Min 117.7ms Max 244.6ms	Count 108 Avg 111.1ms Min 106.4ms Max 195.4ms
350M	Count 125 Avg 65.0ms Min 23.5ms Max 106.3ms	Count 61 Avg 142.8ms Min 136.4ms Max 258.1ms	Count 67 Avg 121.8ms Min 115.9ms Max 192.4ms
400M	Count 102 Avg 71.2ms Min 59.7ms Max 127.5ms	Count 34 Avg 157.3ms Min 149.8ms Max 271.4ms	Count 48 Avg 130.3ms Min 123.1ms Max 197.1ms
450M	Count 82 Avg 67.7ms Min 59.9ms Max 112.7ms	Count 24 Avg 168.7ms Min 159.3ms Max 279.2ms	Count 38 Avg 137.0ms Min 128.5ms Max 197.3ms
500M	Count 69 Avg 66.0ms Min 59.8ms Max 132.1ms	Count 18 Avg 178.3ms Min 168.6ms Max 287.9ms	Count 31 Avg 142.7ms Min 133.7ms Max 196.7ms
550M	Count 61 Avg 69.1ms Min 59.8ms Max 132.8ms	Count 14 Avg 187.5ms Min 175.9ms Max 290.4ms	Count 26 Avg 147.2ms Min 136.6ms Max 196.3ms
600M	Count 53 Avg 67.6ms Min 59.8ms Max 131.7ms	Count 12 Avg 195.8ms Min 182.8ms Max 298.8ms	⁄

Table 5.1: Collection Pauses in Copy

Heap Size	btrees	gcold	splay
250M	⁄	⁄	Count 310 Avg 172.4ms Min 162.2ms Max 178.3ms
300M	⁄	Count 275 Avg 361.3ms Min 310.3ms Max 385.9ms	Count 120 Avg 189.2ms Min 185.8ms Max 192.6ms
350M	Count 125 Avg 241.3ms Min 139.7ms Max 263.1ms	Count 61 Avg 369.7ms Min 328.6ms Max 391.8ms	Count 74 Avg 203.8ms Min 196.5ms Max 206.6ms
400M	Count 102 Avg 265.5ms Min 192.4ms Max 318.6ms	Count 34 Avg 379.0ms Min 339.3ms Max 401.1ms	Count 54 Avg 218.7ms Min 208.7ms Max 221.2ms
450M	Count 82 Avg 278.5ms Min 247.2ms Max 327.1ms	Count 24 Avg 387.1ms Min 351.2ms Max 408.1ms	Count 42 Avg 233.2ms Min 219.8ms Max 236.5ms
500M	Count 69 Avg 295.2ms Min 284.0ms Max 310.8ms	Count 18 Avg 396.0ms Min 363.0ms Max 418.7ms	Count 35 Avg 247.1ms Min 232.1ms Max 250.2ms
550M	Count 61 Avg 315.8ms Min 302.5ms Max 362.7ms	Count 14 Avg 403.9ms Min 372.5ms Max 423.6ms	Count 29 Avg 261.5ms Min 245.0ms Max 264.9ms
600M	Count 53 Avg 331.5ms Min 320.3ms Max 359.7ms	Count 12 Avg 413.4ms Min 382.9ms Max 434.2ms	⁄

Table 5.2: Collection Pauses in Compact

Phase	350M	400M	450M	500M	550M	600M
Full Total	Count 16 Avg 187.8ms Min 71.9ms Max 235.3ms	Count 1 Avg 119.4ms Min 117.7ms Max 121.7ms	Count 1 Avg 166.5ms Min 164.6ms Max 169.1ms	Count 1 Avg 209.3ms Min 204.7ms Max 214.4ms	Count 1 Avg 250.1ms Min 247.6ms Max 252.8ms	∅
Full Marking	Avg 74.1ms Min 11.1ms Max 100.3ms	Avg 31.9ms Min 31.0ms Max 33.2ms	Avg 52.3ms Min 50.9ms Max 53.5ms	Avg 70.3ms Min 68.0ms Max 72.6ms	Avg 87.6ms Min 85.9ms Max 89.4ms	∅
Full Compute Forward	Avg 46.7ms Min 41.2ms Max 49.7ms	Avg 49.8ms Min 48.5ms Max 52.0ms	Avg 56.9ms Min 55.7ms Max 57.9ms	Avg 64.6ms Min 63.1ms Max 65.3ms	Avg 71.5ms Min 70.0ms Max 72.2ms	∅
Full Update References	Avg 32.8ms Min 4.7ms Max 45.6ms	Avg 13.2ms Min 12.9ms Max 13.7ms	Avg 22.6ms Min 21.4ms Max 25.3ms	Avg 31.2ms Min 28.5ms Max 34.0ms	Avg 38.7ms Min 37.0ms Max 41.2ms	∅
Full Relocate	Avg 29.2ms Min 5.4ms Max 41.4ms	Avg 15.3ms Min 14.8ms Max 16.2ms	Avg 25.5ms Min 24.2ms Max 26.6ms	Avg 33.9ms Min 32.4ms Max 35.8ms	Avg 42.8ms Min 40.5ms Max 46.9ms	∅
Full Reset Cards	Avg 0.0ms Min 0.0ms Max 0.1ms	Avg 0.1ms Min 0.0ms Max 0.1ms	Avg 0.1ms Min 0.1ms Max 0.1ms	Avg 0.1ms Min 0.1ms Max 0.1ms	Avg 0.1ms Min 0.1ms Max 0.1ms	∅
Minor Total	Count 579 Avg 7.6ms Min 0.2ms Max 141.6ms	Count 268 Avg 5.1ms Min 0.2ms Max 158.2ms	Count 225 Avg 4.1ms Min 0.2ms Max 176.7ms	Count 2471 Avg 1.2ms Min 0.6ms Max 198.7ms	Count 808 Avg 2.6ms Min 0.6ms Max 217.2ms	Count 225 Avg 4.8ms Min 0.4ms Max 236.8ms
Minor Roots	Avg 0.3ms Min 0.0ms Max 1.1ms	Avg 0.2ms Min 0.0ms Max 0.9ms	Avg 0.2ms Min 0.0ms Max 1.4ms	Avg 0.6ms Min 0.0ms Max 1.5ms	Avg 0.6ms Min 0.0ms Max 1.2ms	Avg 0.4ms Min 0.0ms Max 0.9ms
Minor Tracing	Avg 7.3ms Min 0.0ms Max 141.6ms	Avg 4.9ms Min 0.0ms Max 158.2ms	Avg 3.8ms Min 0.0ms Max 176.7ms	Avg 0.6ms Min 0.0ms Max 198.7ms	Avg 2.1ms Min 0.0ms Max 217.2ms	Avg 4.4ms Min 0.0ms Max 236.8ms

Table 5.3: Collection Pauses in *binarytrees* with Swiper

Phase	300M	350M	400M	450M	500M	550M	600M
Full Total	Count 76 Avg 434.4ms Min 300.5ms Max 478.5ms	Count 15 Avg 463.7ms Min 372.1ms Max 505.2ms	Count 8 Avg 485.2ms Min 414.0ms Max 529.5ms	Count 5 Avg 500.9ms Min 451.9ms Max 534.7ms	Count 4 Avg 521.0ms Min 478.8ms Max 556.6ms	Count 3 Avg 532.4ms Min 501.7ms Max 563.6ms	Count 2 Avg 540.6ms Min 521.4ms Max 564.9ms
Full Marking	Avg 256.0ms Min 146.9ms Max 288.4ms	Avg 264.4ms Min 189.4ms Max 296.6ms	Avg 268.3ms Min 212.7ms Max 300.7ms	Avg 271.1ms Min 232.5ms Max 295.8ms	Avg 276.4ms Min 245.3ms Max 300.4ms	Avg 277.6ms Min 253.0ms Max 300.6ms	Avg 276.9ms Min 260.8ms Max 293.3ms
Full Compute Forward	Avg 44.3ms Min 42.6ms Max 45.3ms	Avg 52.9ms Min 50.1ms Max 54.2ms	Avg 60.7ms Min 58.7ms Max 62.3ms	Avg 67.5ms Min 65.4ms Max 68.9ms	Avg 74.3ms Min 72.5ms Max 76.1ms	Avg 80.8ms Min 79.3ms Max 82.5ms	Avg 86.8ms Min 85.4ms Max 88.6ms
Full Update References	Avg 74.6ms Min 52.4ms Max 86.8ms	Avg 80.0ms Min 65.1ms Max 92.9ms	Avg 84.1ms Min 71.8ms Max 95.0ms	Avg 86.4ms Min 76.5ms Max 94.9ms	Avg 89.5ms Min 81.6ms Max 98.0ms	Avg 91.8ms Min 85.9ms Max 98.3ms	Avg 92.2ms Min 87.8ms Max 97.6ms
Full Relocate	Avg 59.1ms Min 54.6ms Max 64.1ms	Avg 64.4ms Min 59.0ms Max 68.0ms	Avg 68.6ms Min 62.9ms Max 71.8ms	Avg 70.9ms Min 65.7ms Max 75.1ms	Avg 74.5ms Min 68.0ms Max 81.5ms	Avg 74.4ms Min 70.3ms Max 77.2ms	Avg 75.7ms Min 72.3ms Max 78.0ms
Full Reset Cards	Avg 0.0ms Min 0.0ms Max 0.1ms	Avg 0.0ms Min 0.0ms Max 0.1ms	Avg 0.1ms Min 0.0ms Max 0.1ms	Avg 0.1ms Min 0.1ms Max 0.1ms	Avg 0.1ms Min 0.1ms Max 0.1ms	Avg 0.1ms Min 0.1ms Max 0.1ms	Avg 0.1ms Min 0.1ms Max 0.1ms
Minor Total	Count 1732 Avg 1.7ms Min 0.5ms Max 117.7ms	Count 600 Avg 4.2ms Min 0.6ms Max 141.2ms	Count 378 Avg 6.4ms Min 0.7ms Max 156.8ms	Count 269 Avg 8.9ms Min 0.7ms Max 176.8ms	Count 220 Avg 10.7ms Min 0.8ms Max 197.2ms	Count 178 Avg 13.2ms Min 0.8ms Max 216.4ms	Count 151 Avg 15.6ms Min 0.9ms Max 235.3ms
Minor Roots	Avg 0.7ms Min 0.0ms Max 1.7ms	Avg 1.4ms Min 0.0ms Max 3.8ms	Avg 1.9ms Min 0.0ms Max 8.7ms	Avg 2.4ms Min 0.0ms Max 7.1ms	Avg 2.7ms Min 0.0ms Max 8.6ms	Avg 3.1ms Min 0.0ms Max 9.8ms	Avg 3.6ms Min 0.0ms Max 11.0ms
Minor Tracing	Avg 0.9ms Min 0.2ms Max 117.7ms	Avg 2.8ms Min 0.2ms Max 141.2ms	Avg 4.6ms Min 0.2ms Max 156.8ms	Avg 6.5ms Min 0.2ms Max 176.8ms	Avg 8.0ms Min 0.2ms Max 197.2ms	Avg 10.1ms Min 0.1ms Max 216.3ms	Avg 12.0ms Min 0.2ms Max 235.3ms

Table 5.4: Collection Pauses in *gcold* with Swiper

Phase	250M	300M	350M	400M	450M	500M	550M
Full Total	Count 231 Avg 211.5ms Min 196.4ms Max 216.8ms	Count 86 Avg 227.7ms Min 216.5ms Max 233.4ms	Count 53 Avg 241.0ms Min 229.7ms Max 248.1ms	Count 38 Avg 252.7ms Min 240.4ms Max 262.2ms	Count 30 Avg 263.0ms Min 256.7ms Max 270.2ms	Count 25 Avg 273.7ms Min 264.6ms Max 289.0ms	Count 21 Avg 285.6ms Min 277.8ms Max 299.9ms
Full Marking	Avg 106.8ms Min 89.1ms Max 111.1ms	Avg 107.9ms Min 91.8ms Max 112.4ms	Avg 108.3ms Min 94.5ms Max 114.0ms	Avg 108.9ms Min 98.0ms Max 115.4ms	Avg 108.7ms Min 99.1ms Max 115.2ms	Avg 109.2ms Min 103.5ms Max 116.2ms	Avg 110.4ms Min 102.9ms Max 118.0ms
Full Compute Forward	Avg 29.3ms Min 28.9ms Max 30.1ms	Avg 34.5ms Min 33.9ms Max 35.7ms	Avg 39.5ms Min 38.9ms Max 41.0ms	Avg 44.6ms Min 43.3ms Max 48.3ms	Avg 49.3ms Min 48.4ms Max 52.4ms	Avg 54.2ms Min 53.1ms Max 57.7ms	Avg 59.4ms Min 57.9ms Max 63.1ms
Full Update References	Avg 34.8ms Min 34.2ms Max 37.7ms	Avg 39.1ms Min 38.3ms Max 42.5ms	Avg 42.5ms Min 41.4ms Max 46.6ms	Avg 45.1ms Min 43.6ms Max 52.0ms	Avg 47.4ms Min 45.9ms Max 52.5ms	Avg 49.5ms Min 47.4ms Max 56.7ms	Avg 51.6ms Min 49.6ms Max 58.9ms
Full Relocate	Avg 39.5ms Min 37.6ms Max 41.9ms	Avg 43.7ms Min 42.0ms Max 47.1ms	Avg 46.7ms Min 44.8ms Max 51.1ms	Avg 48.9ms Min 46.9ms Max 54.7ms	Avg 51.0ms Min 48.6ms Max 57.2ms	Avg 52.9ms Min 50.9ms Max 61.2ms	Avg 55.0ms Min 52.3ms Max 63.6ms
Full Reset Cards	Avg 0.0ms Min 0.0ms Max 0.1ms	Avg 0.0ms Min 0.0ms Max 0.1ms	Avg 0.0ms Min 0.0ms Max 0.1ms	Avg 0.1ms Min 0.0ms Max 0.1ms	Avg 0.1ms Min 0.1ms Max 0.1ms	Avg 0.1ms Min 0.1ms Max 0.1ms	Avg 0.1ms Min 0.1ms Max 0.1ms
Minor Total	Count 3366 Avg 3.1ms Min 0.7ms Max 82.8ms	Count 1726 Avg 5.9ms Min 0.8ms Max 98.8ms	Count 1197 Avg 8.4ms Min 0.9ms Max 116.0ms	Count 925 Avg 10.9ms Min 0.9ms Max 132.8ms	Count 760 Avg 13.3ms Min 1.0ms Max 152.3ms	Count 659 Avg 15.3ms Min 1.1ms Max 170.3ms	Count 573 Avg 17.6ms Min 1.1ms Max 186.2ms
Minor Roots	Avg 0.8ms Min 0.0ms Max 2.1ms	Avg 1.0ms Min 0.0ms Max 2.5ms	Avg 1.2ms Min 0.0ms Max 3.1ms	Avg 1.3ms Min 0.0ms Max 3.7ms	Avg 1.4ms Min 0.0ms Max 4.4ms	Avg 1.5ms Min 0.0ms Max 4.5ms	Avg 1.7ms Min 0.0ms Max 4.7ms
Minor Tracing	Avg 2.3ms Min 0.3ms Max 82.8ms	Avg 4.9ms Min 0.3ms Max 98.8ms	Avg 7.3ms Min 0.3ms Max 116.0ms	Avg 9.6ms Min 0.3ms Max 132.8ms	Avg 11.8ms Min 0.3ms Max 152.3ms	Avg 13.7ms Min 0.3ms Max 170.3ms	Avg 15.9ms Min 0.3ms Max 186.1ms

Table 5.5: Collection Pauses in *splay* with Swiper

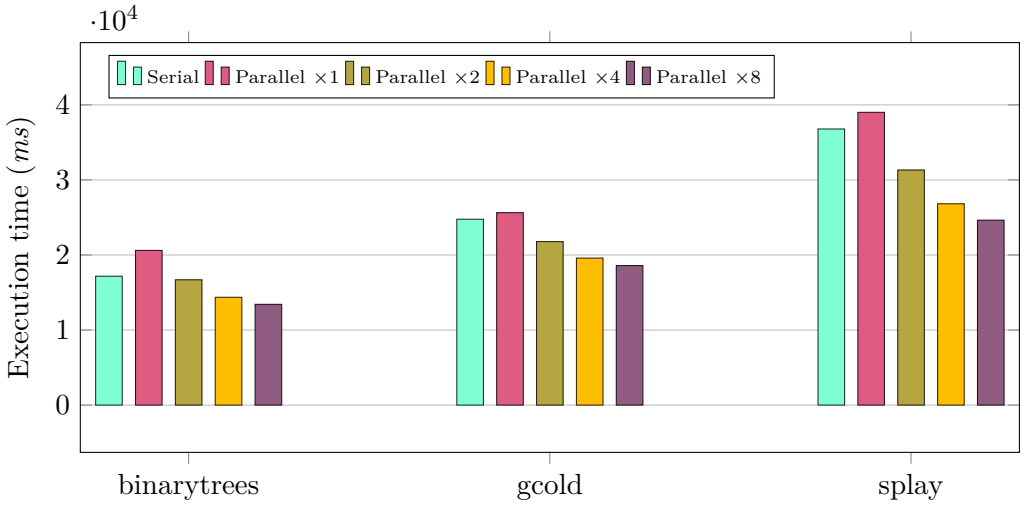

Figure 5.16: Total Time with Parallel Swiper

5.4 Parallel Swiper

This section compares *Swiper*'s parallel collections to their serial version. Benchmarks are run with 1, 2, 4 and 8 worker threads in the parallel collection. This time benchmarks are executed with exactly one heap size each: *binarytrees* and *gcold* with 350M, while *splay* uses 400M. Heap size was selected such that many collections need to be performed during execution. Results are shown in the figures 5.16-5.20. The individual collection phases are listed in the tables 5.6-5.8.

When using parallel collection, mutator time increases slightly. This is due to reduced locality, resulting from a different order of copying in the minor collection. Parallel collection with one worker thread is less efficient than the serial collection. This is as anticipated, because there is some overhead and synchronization needed for parallel collections. Whereas collection time with two worker threads already improves on the serial collection in these benchmarks. Increasing the number of threads even reduces collection time further. The improvement in collection time is large enough such that total execution time improves as well.

Using work-stealing for parallel minor collections also introduces non-determinism in the collection. Allocating objects in LABs might increase memory usage in the young and old generation and might cause more or sooner collections than the serial version. This could also vary in each run. Nevertheless both pause times for minor and full collections improve using multiple worker threads significantly. Reducing minor collections pause times is harder since most collections are already quite short. The effect of parallelization is here best seen in the maximum minor collection pause time. Note that *reset cards* is merged with the *relocation* phase in the parallel implementation.

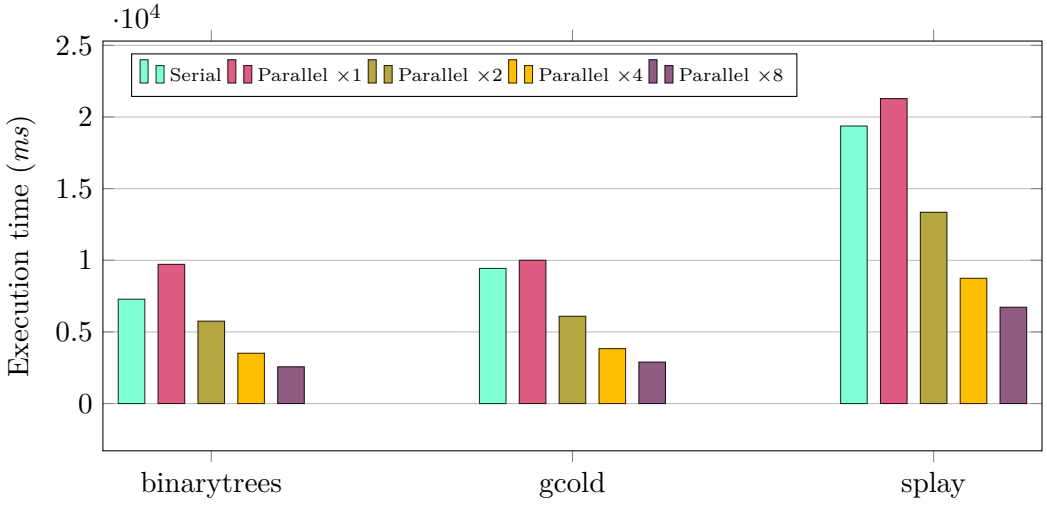

Figure 5.17: Collection Time with Parallel Swiper

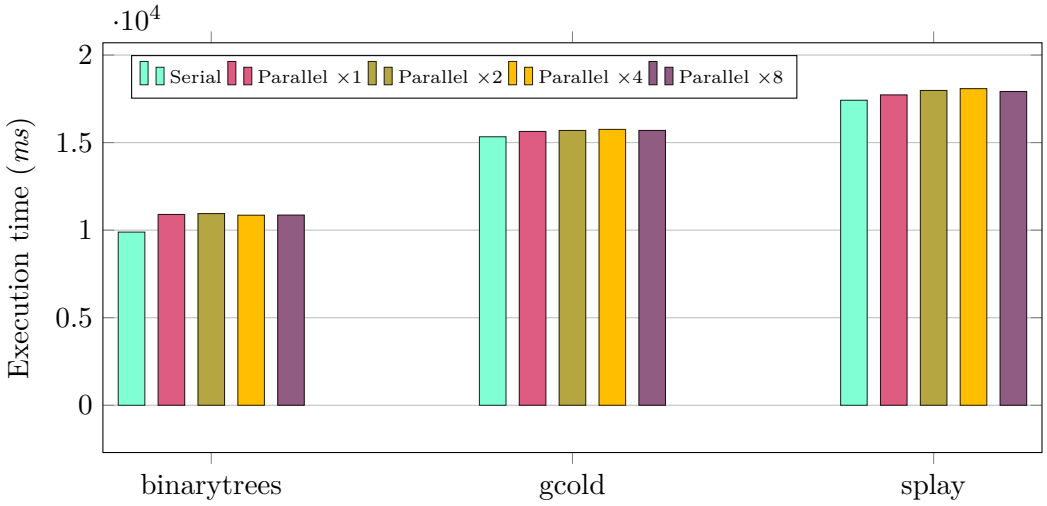

Figure 5.18: Mutator Time with Parallel Swiper

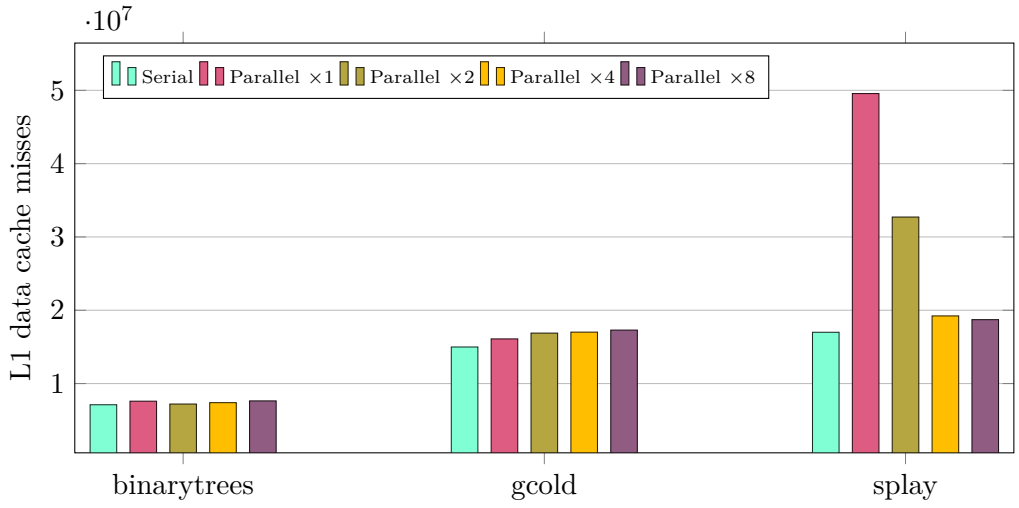

Figure 5.19: L1 data cache misses with Parallel Swiper

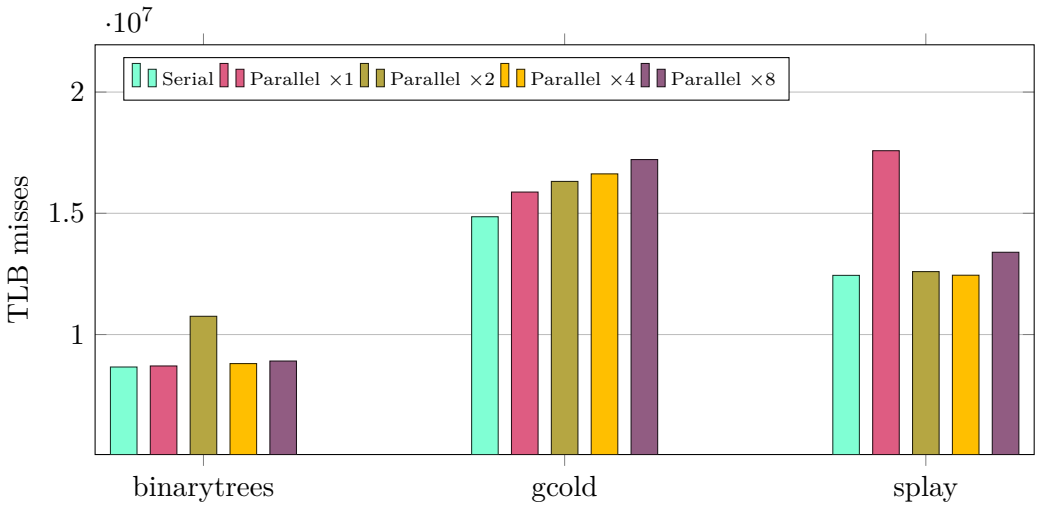

Figure 5.20: TLB misses with Parallel Swiper

Phase	Serial	Parallel ×1	Parallel ×2	Parallel ×4	Parallel ×8
Full Total	Count 16 Avg 187.8ms Min 72.7ms Max 238.4ms	Count 16 Avg 218.9ms Min 79.8ms Max 235.4ms	Count 16 Avg 140.3ms Min 52.5ms Max 165.9ms	Count 16 Avg 89.8ms Min 35.3ms Max 108.0ms	Count 16 Avg 67.9ms Min 27.4ms Max 79.1ms
Full Marking	Avg 74.5ms Min 11.1ms Max 102.1ms	Avg 68.8ms Min 10.7ms Max 79.8ms	Avg 36.9ms Min 6.0ms Max 45.1ms	Avg 20.7ms Min 3.3ms Max 25.3ms	Avg 12.5ms Min 1.8ms Max 16.8ms
Full Compute Forward	Avg 46.6ms Min 41.5ms Max 49.5ms	Avg 66.0ms Min 45.0ms Max 72.8ms	Avg 37.7ms Min 23.8ms Max 40.7ms	Avg 22.8ms Min 13.2ms Max 25.8ms	Avg 19.5ms Min 10.2ms Max 22.1ms
Full Update References	Avg 32.3ms Min 4.7ms Max 46.2ms	Avg 47.7ms Min 7.4ms Max 54.2ms	Avg 27.0ms Min 5.8ms Max 32.0ms	Avg 18.1ms Min 3.2ms Max 22.4ms	Avg 13.6ms Min 2.2ms Max 16.7ms
Full Relocate	Avg 29.4ms Min 5.4ms Max 41.7ms	Avg 29.5ms Min 5.3ms Max 37.7ms	Avg 32.0ms Min 5.2ms Max 43.6ms	Avg 21.4ms Min 4.4ms Max 29.7ms	Avg 16.5ms Min 2.9ms Max 20.7ms
Full Reset Cards	Avg 0.0ms Min 0.0ms Max 0.1ms	Avg 0.0ms Min 0.0ms Max 0.0ms	Avg 0.0ms Min 0.0ms Max 0.0ms	Avg 0.0ms Min 0.0ms Max 0.0ms	Avg 0.0ms Min 0.0ms Max 0.0ms
Minor Total	Count 579 Avg 7.4ms Min 0.2ms Max 137.4ms	Count 584 Avg 10.6ms Min 0.6ms Max 178.0ms	Count 584 Avg 6.0ms Min 0.4ms Max 97.3ms	Count 581 Avg 3.6ms Min 0.3ms Max 56.4ms	Count 574 Avg 2.6ms Min 0.2ms Max 40.6ms
Minor Roots	Avg 0.3ms Min 0.0ms Max 0.9ms	Avg 1.0ms Min 0.0ms Max 2.1ms	Avg 0.6ms Min 0.1ms Max 1.1ms	Avg 0.4ms Min 0.1ms Max 0.6ms	Avg 0.3ms Min 0.1ms Max 1.1ms
Minor Tracing	Avg 7.1ms Min 0.0ms Max 137.4ms	Avg 9.6ms Min 0.0ms Max 177.9ms	Avg 5.4ms Min 0.0ms Max 97.2ms	Avg 3.2ms Min 0.0ms Max 56.3ms	Avg 2.3ms Min 0.0ms Max 40.5ms

Table 5.6: Collection Pauses in *binarytrees* with Parallel Swiper

Phase	Serial	Parallel ×1	Parallel ×2	Parallel ×4	Parallel ×8
Full Total	Count 15 Avg 463.6ms Min 371.4ms Max 506.2ms	Count 15 Avg 411.7ms Min 359.5ms Max 444.0ms	Count 15 Avg 252.2ms Min 223.6ms Max 269.6ms	Count 15 Avg 158.3ms Min 136.9ms Max 171.2ms	Count 16 Avg 117.4ms Min 103.4ms Max 126.4ms
Full Marking	Avg 264.9ms Min 190.8ms Max 296.3ms	Avg 167.1ms Min 131.2ms Max 191.5ms	Avg 88.3ms Min 69.6ms Max 101.2ms	Avg 53.1ms Min 40.9ms Max 60.1ms	Avg 33.4ms Min 24.7ms Max 38.4ms
Full Compute Forward	Avg 52.8ms Min 50.4ms Max 54.1ms	Avg 88.7ms Min 85.7ms Max 91.9ms	Avg 53.1ms Min 49.0ms Max 55.2ms	Avg 32.7ms Min 29.8ms Max 34.6ms	Avg 29.1ms Min 25.7ms Max 31.3ms
Full Update References	Avg 79.9ms Min 63.5ms Max 92.4ms	Avg 94.1ms Min 83.3ms Max 102.3ms	Avg 50.0ms Min 44.5ms Max 54.9ms	Avg 32.3ms Min 29.5ms Max 34.8ms	Avg 23.0ms Min 21.4ms Max 24.8ms
Full Relocate	Avg 64.1ms Min 58.7ms Max 66.8ms	Avg 59.2ms Min 56.1ms Max 66.1ms	Avg 58.0ms Min 53.0ms Max 65.5ms	Avg 37.4ms Min 34.1ms Max 43.3ms	Avg 29.5ms Min 27.6ms Max 33.3ms
Full Reset Cards	Avg 0.0ms Min 0.0ms Max 0.1ms	Avg 0.0ms Min 0.0ms Max 0.0ms	Avg 0.0ms Min 0.0ms Max 0.0ms	Avg 0.0ms Min 0.0ms Max 0.0ms	Avg 0.0ms Min 0.0ms Max 0.0ms
Minor Total	Count 600 Avg 4.1ms Min 0.6ms Max 135.9ms	Count 594 Avg 6.4ms Min 1.4ms Max 174.5ms	Count 583 Avg 4.0ms Min 0.8ms Max 95.0ms	Count 570 Avg 2.6ms Min 0.6ms Max 55.8ms	Count 561 Avg 1.8ms Min 0.4ms Max 47.3ms
Minor Roots	Avg 1.4ms Min 0.0ms Max 3.8ms	Avg 2.7ms Min 0.1ms Max 7.4ms	Avg 1.7ms Min 0.1ms Max 4.5ms	Avg 1.1ms Min 0.1ms Max 3.0ms	Avg 0.7ms Min 0.1ms Max 2.1ms
Minor Tracing	Avg 2.7ms Min 0.2ms Max 135.9ms	Avg 3.8ms Min 0.2ms Max 174.3ms	Avg 2.3ms Min 0.2ms Max 94.8ms	Avg 1.5ms Min 0.2ms Max 55.6ms	Avg 1.1ms Min 0.1ms Max 47.1ms

Table 5.7: Collection Pauses in *gcold* with Parallel Swiper

Phase	Serial	Parallel ×1	Parallel ×2	Parallel ×4	Parallel ×8
Full Total	Count 38 Avg 253.4ms Min 241.9ms Max 260.7ms	Count 38 Avg 250.5ms Min 246.0ms Max 284.8ms	Count 39 Avg 165.3ms Min 157.7ms Max 200.5ms	Count 39 Avg 114.5ms Min 109.3ms Max 148.5ms	Count 39 Avg 95.7ms Min 90.7ms Max 158.5ms
Full Marking	Avg 109.3ms Min 98.5ms Max 115.3ms	Avg 67.6ms Min 66.7ms Max 72.5ms	Avg 37.7ms Min 36.9ms Max 39.0ms	Avg 23.5ms Min 23.0ms Max 24.0ms	Avg 15.5ms Min 14.9ms Max 16.5ms
Full Compute Forward	Avg 44.8ms Min 43.6ms Max 46.8ms	Avg 78.8ms Min 76.3ms Max 92.2ms	Avg 48.6ms Min 45.5ms Max 66.2ms	Avg 32.8ms Min 29.5ms Max 56.2ms	Avg 33.6ms Min 30.0ms Max 90.8ms
Full Update References	Avg 44.9ms Min 43.4ms Max 50.7ms	Avg 51.1ms Min 50.1ms Max 60.3ms	Avg 28.9ms Min 28.2ms Max 34.6ms	Avg 21.3ms Min 20.8ms Max 23.9ms	Avg 16.0ms Min 15.3ms Max 18.5ms
Full Relocate	Avg 49.1ms Min 46.9ms Max 54.6ms	Avg 46.2ms Min 45.2ms Max 58.4ms	Avg 43.6ms Min 38.4ms Max 58.2ms	Avg 30.3ms Min 26.7ms Max 39.3ms	Avg 24.8ms Min 23.2ms Max 29.0ms
Full Reset Cards	Avg 0.1ms Min 0.0ms Max 0.1ms	Avg 0.0ms Min 0.0ms Max 0.0ms	Avg 0.0ms Min 0.0ms Max 0.0ms	Avg 0.0ms Min 0.0ms Max 0.0ms	Avg 0.0ms Min 0.0ms Max 0.0ms
Minor Total	Count 925 Avg 10.5ms Min 0.9ms Max 132.3ms	Count 930 Avg 12.7ms Min 1.9ms Max 134.3ms	Count 936 Avg 7.4ms Min 1.1ms Max 74.8ms	Count 924 Avg 4.6ms Min 0.7ms Max 45.8ms	Count 905 Avg 3.3ms Min 0.5ms Max 33.1ms
Minor Roots	Avg 1.2ms Min 0.0ms Max 3.8ms	Avg 2.3ms Min 0.0ms Max 5.9ms	Avg 1.4ms Min 0.1ms Max 3.4ms	Avg 0.9ms Min 0.1ms Max 1.9ms	Avg 0.6ms Min 0.2ms Max 3.2ms
Minor Tracing	Avg 9.3ms Min 0.3ms Max 132.2ms	Avg 10.4ms Min 0.4ms Max 134.1ms	Avg 6.0ms Min 0.3ms Max 74.6ms	Avg 3.8ms Min 0.2ms Max 45.5ms	Avg 2.7ms Min 0.1ms Max 32.7ms

Table 5.8: Collection Pauses in *splay* with Parallel Swiper

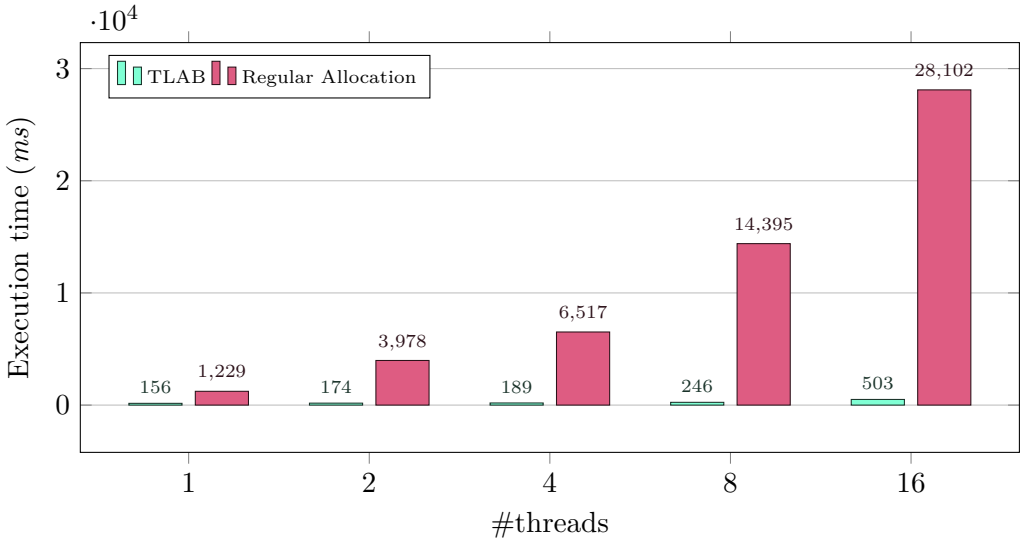

Figure 5.21: TLAB Allocation vs. Regular Allocation *(1)*

5.5 Allocation Time

The next benchmark allocates 20 million objects in a tight loop in each started thread. The benchmark is run using the *zero* collector with a large enough heap. *Zero* is used because this benchmark focuses solely on the allocation time of the runtime using TLABs. While Dora is single-threaded at the moment, it already supports multiple threads partially. The current implementation allows to test TLAB allocation on multiple threads.

The results show that TLABs decrease allocation time substantially in figures 5.22 and 5.21. The difference between these two approaches becomes bigger with the number of threads allocating. This behavior matches the expectation, since more threads increase contention in the allocation routine. When using TLABs most allocations are still served from the local buffer, contention increases less compared to the regular allocation. Nevertheless even the single-threaded run already needs more than fives times as long when allocating objects with the regular allocation. In addition to synchronization, it is also expensive to call into the runtime compared to the few instructions needed for TLAB allocation.

This improvement in allocation performance is also observable in the micro-benchmarks used. Only *Splunc*'s performance does not depend on allocation as much, while *binarytrees* on the other hand allocates a lot of memory and performance becomes nearly 4 times better thanks to the use of TLABs. When running 16 threads, the operating system needs to schedule the threads onto the 8 CPU cores.

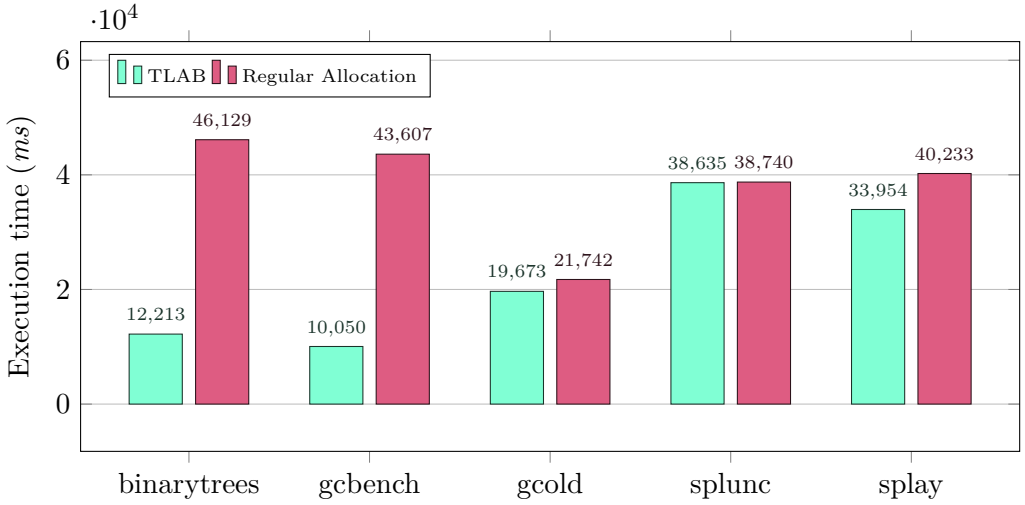

Figure 5.22: TLAB Allocation vs. Regular Allocation *(2)*

5.6 False Sharing

Section 4.3 mentioned that *false sharing* might become a problem due to *Swiper*'s write barrier. Therefore a benchmark was written to determine how much this can influcence throughput. The application initially creates 32K of objects and promotes those objects into the old generation. Immediately afterwards it starts all worker threads, which write into a distinct subset of these objects in the old generation. Each object is modified by exactly one thread.

The benchmark was executed with 1, 2, 4, 8, 16 and 32 threads. The CPU has only 8 cores, so for 16 and 32 threads the operating system has to schedule the threads onto the cores. The execution time with *Swiper* is compared to the *Zero* collector. *Zero* is well suited for this comparison because it does not need a write barrier. To prove that the additional overhead comes indeed from the write barrier, *Swiper* was modified to not emit write barriers. While this does not work in general, it is enough for this short benchmark.

The results in figure 5.23 show that performance becomes worse with the number of running threads. While performance degrades, it is not enough to warrant a different write barrier for now. Also the regression might be fixed when using a conditional store instead, as was discussed previously in 4.3.

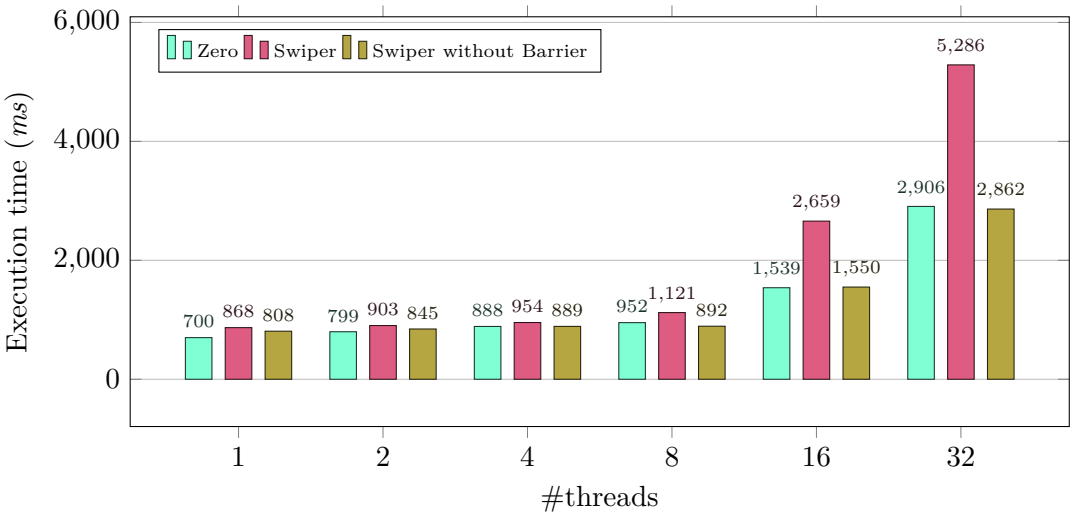

Figure 5.23: False Sharing with Zero and Swiper

CHAPTER 6

Future Work and Summary

This chapter discusses possible directions for future work and will give a summary about this thesis.

6.1 Future Work

While the current implementation of *Swiper* is a fully working generational GC, there is always room for further improvement. Chapter 5 shows that while the GC does its best to keep pauses short, they might still be too long for some use cases. Therefore a future goal for Swiper is to reduce pause times even further while not sacrificing throughput or memory usage too much.

6.1.1 Concurrent Marking

As previously discussed in chapter 5, *marking* is an expensive phase of the *mark-compact* collection. As a consequence it would be worthwhile to move the *marking* phase out of the collection pause. Section 2.4 discussed techniques to achieve this by using either incremental or concurrent marking. In the case of Dora concurrent marking seems to be the most reasonable choice. Dora is typically used on machines with multiple CPU cores. Also there is no marking operation in Dora that would require the application to be paused as well.

Using concurrent marking requires an additional write barrier in the mutator. This certainly reduces throughput of the application, hence this feature should be able to be turned-off at application startup. Making this feature optional should come with no performance disadvantage and is quite useful for testing.

Since the goal for *Swiper* is to be a good general-purpose collector it makes sense to use a gray mutator architecture. Using a *read* barrier for this collector would certainly

considered to be too expensive. Having a gray mutator allows the runtime to allocate objects *white* and generally reduces the amount of floating garbage in a collection. A *Snapshot-at-the-beginning* approach would retain more objects and increases memory usage. *Swiper* should use a slightly changed Dijkstra *write* barrier for concurrent marking similar to V8. It might retain more objects than the Steele barrier but using barrier guarantees that marking makes progress. With the Dijkstra barrier the mutator could still outpace the collector by allocating more and more objects. However with the Steele barrier, the mutator could create more marking work simply by modifying objects. The collector therefore has to make sure it makes progress, which might require additional incremental marking pauses.

The Dijkstra barrier requires to scan the roots at least twice, both at the start and end of the marking phase. While this could increase pause times when discovering a large object graph in the final marking pause, this should be acceptable for *Swiper*. Black mutator collectors face a similar problem when draining write barrier buffers in the final marking pause.

6.1.2 Mark Sweep

While *mark-compact* has some advantages it is also responsible for longer pause times in Dora. *Swiper* reduces pause times by using multiple threads and performing work in parallel. Nevertheless pause time will always be longer since objects are relocated and the heap has to be scanned multiple times. For very large heaps full collection pause times could take multiple seconds, which might be too long in specific use cases - even when such pauses should be quite rare.

Switching the full collection to *mark-sweep* would decrease pause times significantly. The *marking* phase can run concurrently to the application just like with *mark-compact*. Sweeping the heap can easily run in a separate thread outside the collection pause as well. Changing the collection scheme requires major adjustments to the collector though. This is why this feature probably cannot be offered as another option of *Swiper*. It will probably have to be implemented in a separate collector and reuse code from *Swiper*. Using *mark-sweep* for the old generation also necessitates changes to the minor collection. During such collections objects are promoted into the old generation. With *mark-sweep* objects cannot be promoted anymore simply by using bump-pointer allocation. Objects are then segregated by size and allocated in blocks. The *crossing map* used to find the first object in a card when using *mark-compact* could be removed. Another improvement possible with *mark-sweep* would be to get rid of the second word in each object's header. It currently is used to store the forwarding pointer during a full collection.

While bump-pointer allocation is not possible with *mark-sweep* this should not regress performance too much since sequential allocation is still used in the young generation. Objects are only allocated in the old generation during collections. This is also the reason allocation locality is also believed to become only slightly worse than with *mark-compact*. Therefore *mark-sweep* should bring shorter pause times and smaller object headers while

regressing other characteristics of *Swiper* only slightly. Many applications should actually benefit from this change.

6.1.3 Concurrent Compaction

While *Swiper* is meant to be used us as a general-purpose collector, it makes sense to add another collector trying to minimize latency as much as possible. There are real-time usages where *Swiper*'s pause times would be insufficient even with both concurrent marking and *mark-sweep* in the old generation. Trying to reduce pause times even further with *Swiper* would probably decrease throughput too much. Especially considering that many applications do not have such rigid latency requirements. Therefore it makes sense for Dora to have a separate collector that focuses on latency above all else.

This new collector would ideally still be moving objects such that bump-pointer allocation is still possible. However the relocation of objects would be performed concurrently to the application. Pauses would only perform work proportional to the root set size. The collector splits the heap into regions of equal size. Marking and evacuation of objects should be performed in a single combined phase. In an initial pause to start the collection cycle, the collector chooses a collection set based on the liveness information of the last collection's cycle. In this initial pause the collector would therefore also mark and evacuate objects referenced from the root set. Afterwards concurrent GC worker threads perform marking and evacuation concurrently to the application. When the GC threads run out of work, the application is stopped again to finish the collection cycle. Evacuated regions can be freed at this point. Scanning the root set is only necessary in the initial pause since this collector would require a black mutator and as a consequence black allocation.

In this algorithm the *read* barrier would check the color of an object. If the object was still white, it is marked and if part of the collection set then also evacuated. This ensures that the mutator only sees objects that were already marked and evacuated.

6.1.4 Stop the World

While the runtime and all the collectors would already support multi-threading, there is one operation left to be implemented: *stopping the world*. This operation stops all application threads such that the GC can collect garbage without any interference from the mutator. In a single-threaded application this is not necessary since the collector is always invoked through the allocation function. Simply invoking the runtime from the sole application thread conceptually stops the mutator. In contrast to that, in a multi-threaded system the collector might be invoked when one application thread runs out of memory. However all the other threads might continue to run, either because they do not allocate objects at the moment or have sufficient thread-local storage. Therefore the runtime needs a way to stop application threads. Dora already emits additional code at the end of functions and all loops that check whether a pause was requested from another thread. In the case there was, the thread calls into the runtime to block

its execution. Placing the checks in the generated code guarantees that all threads are eventually suspended.

Nevertheless this does not solve the problem yet. These checks are only inserted into managed code written in Dora. However managed code might still invoke native code written in another programming language like Rust or C. An application thread that executes native code at the moment a GC is requested does not have to be stopped. The thread is only suspended when the native code tries to access the heap or returns into managed code before the pause is over. This requires adapting Dora's *entry* and *exit* stubs, they are executed before entering and after leaving native code.

6.2 Summary

This thesis discussed *Swiper*, a generational and parallel tracing garbage collector. It is written in Rust and is used in the Dora runtime for memory management. The main goal was to have good throughput with acceptable pause times.

To this end the heap was split into *young* and *old* generation. Objects are allocated by the default in the *young* generation and are promoted when they become old enough. This organization is based on the *weak generational hypothesis* that states that most objects die young. This means that the collector is more effective when focusing collection on areas that are likely to reclaim more memory: the *young* generation. The collector therefore implements two kind of collections: *minor* and *full* collection. Typically *Swiper* frequently performs *minor* collections that collect garbage only in the *young* generation. Infrequent *full* collections are required to reclaim memory for the whole heap. Parallelizing the collection decreases pause times by distributing work to multiple threads. Using moving collection schemes allows the mutator to use the efficient bump-pointer allocation.

Bibliography

[ABCS03] Clement R. Attanasio, David F. Bacon, Anthony Cocchi, and Stephen Smith. A comparative evaluation of parallel garbage collector implementations. In *Proceedings of the 14th International Conference on Languages and Compilers for Parallel Computing*, LCPC'01, pages 177–192, Berlin, Heidelberg, 2003. Springer-Verlag.

[Age98] Ole Agesen. Gc points in a threaded environment. 1998.

[ARM18] ARM Limited. *ARMv8 Architecture Reference Manual*. October 2018.

[Bak78] Henry G. Baker, Jr. List processing in real time on a serial computer. *Commun. ACM*, 21(4):280–294, April 1978.

[BCM04] Stephen M. Blackburn, Perry Cheng, and Kathryn S. McKinley. Oil and water? high performance garbage collection in java with mmtk. In *Proceedings of the 26th International Conference on Software Engineering*, ICSE '04, pages 137–146, Washington, DC, USA, 2004. IEEE Computer Society.

[Bro84] Rodney A. Brooks. Trading data space for reduced time and code space in real-time garbage collection on stock hardware. In *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming*, LFP '84, pages 256–262, New York, NY, USA, 1984. ACM.

[CH93] Urs Hlzle Computer and Urs Hölzle. A fast write barrier for generational garbage collectors. In *OOPSLA/ECOOP '93 Workshop on Garbage Collection in Object-Oriented Systems*, 1993.

[Che70] C. J. Cheney. A nonrecursive list compacting algorithm. *Commun. ACM*, 13(11):677–678, November 1970.

[CL05] David Chase and Yossi Lev. Dynamic circular work-stealing deque. In *Proceedings of the Seventeenth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '05, pages 21–28, New York, NY, USA, 2005. ACM.

[CUL89] C. Chambers, D. Ungar, and E. Lee. An efficient implementation of self a dynamically-typed object-oriented language based on prototypes. In *Conference Proceedings on Object-oriented Programming Systems, Languages and Applications*, OOPSLA '89, pages 49–70, New York, NY, USA, 1989. ACM.

[DEE+16] Ulan Degenbaev, Jochen Eisinger, Manfred Ernst, Ross McIlroy, and Hannes Payer. Idle time garbage collection scheduling. *SIGPLAN Not.*, 51(6):570–583, June 2016.

[DLM+78] Edsger W. Dijkstra, Leslie Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. On-the-fly garbage collection: An exercise in cooperation. *Commun. ACM*, 21(11):966–975, November 1978.

[FCJH16] Henrique Ferreiro, Laura Castro, Vladimir Janjic, and Kevin Hammond. Kindergarten cop: Dynamic nursery resizing for ghc. In *Proceedings of the 25th International Conference on Compiler Construction*, CC 2016, pages 56–66, New York, NY, USA, 2016. ACM.

[FDSZ01] Christine H Flood, David Detlefs, Nir Shavit, and Xiolan Zhang. Parallel garbage collection for shared memory multiprocessors. In *Java Virtual Machine Research and Technology Symposium*, 2001.

[FKD+16] Christine H. Flood, Roman Kennke, Andrew Dinn, Andrew Haley, and Roland Westrelin. 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*, PPPJ '16, pages 13:1–13:9, New York, NY, USA, 2016. ACM.

[Hal84] Robert H. Halstead, Jr. Implementation of multilisp: Lisp on a multiprocessor. In *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming*, LFP '84, pages 9–17, New York, NY, USA, 1984. ACM.

[Hel12] Erik Helin. Improving load balancing during the marking phase of garbage collection, 2012.

[HJH10] Laurence Hellyer, Richard Jones, and Antony L. Hosking. The locality of concurrent write barriers. In *Proceedings of the 2010 International Symposium on Memory Management*, ISMM '10, pages 83–92, New York, NY, USA, 2010. ACM.

[Hud18] Richard L. Hudson. Getting to Go: The Journey of Go's Garbage Collector. `https://blog.golang.org/ismmkeynote`, 2018. [Online; accessed 19-December-2018].

[Int19] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual*. Number 253669-033US. January 2019.

[JHM11] Richard Jones, Antony Hosking, and Eliot Moss. *The Garbage Collection Handbook: The Art of Automatic Memory Management*. Chapman & Hall/CRC, 1st edition, 2011.

[JL96] Richard Jones and Rafael Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. John Wiley & Sons, Inc., New York, NY, USA, 1996.

[Lid18] Per Liden. The Z Garbage Collector. `https://wiki.openjdk.java.net/display/zgc/Main`, 2018. [Online; accessed 19-December-2018].

[LPCZN13] Nhat Minh Lê, Antoniu Pop, Albert Cohen, and Francesco Zappa Nardelli. Correct and efficient work-stealing for weak memory models. *SIGPLAN Not.*, 48(8):69–80, February 2013.

[LTHB14] Tongping Liu, Chen Tian, Ziang Hu, and Emery D. Berger. Predator: Predictive false sharing detection. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '14, pages 3–14, New York, NY, USA, 2014. ACM.

[Pir98] Pekka P. Pirinen. Barrier techniques for incremental tracing. In *Proceedings of the 1st International Symposium on Memory Management*, ISMM '98, pages 20–25, New York, NY, USA, 1998. ACM.

[Piz17] Filip Pizlo. Introducing Riptide: WebKit's Retreating Wavefront Concurrent Garbage Collector. `https://webkit.org/blog/7122/`, 2017. [Online; accessed 19-December-2018].

[SBH05] Daniel Spoonhower, Guy Blelloch, and Robert Harper. Using page residency to balance tradeoffs in tracing garbage collection. In *Proceedings of the 1st ACM/USENIX International Conference on Virtual Execution Environments*, VEE '05, pages 57–67, New York, NY, USA, 2005. ACM.

[SBM14] Rifat Shahriyar, Stephen M. Blackburn, and Kathryn S. McKinley. Fast conservative garbage collection. *SIGPLAN Not.*, 49(10):121–139, October 2014.

[Ste75] Guy L. Steele, Jr. Multiprocessing compactifying garbage collection. *Commun. ACM*, 18(9):495–508, September 1975.

[Ung84] David Ungar. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. *SIGSOFT Softw. Eng. Notes*, 9(3):157–167, April 1984.

[Wil92] Paul R. Wilson. Uniprocessor garbage collection techniques. In *Proceedings of the International Workshop on Memory Management*, IWMM '92, pages 1–42, London, UK, UK, 1992. Springer-Verlag.

[YBFH12] Xi Yang, Stephen M. Blackburn, Daniel Frampton, and Antony L. Hosking. Barriers reconsidered, friendlier still! *SIGPLAN Not.*, 47(11):37–48, June 2012.

[Yua90] T. Yuasa. Real-time garbage collection on general-purpose machines. *J. Syst. Softw.*, 11(3):181–198, March 1990.