School of Computer Science and Engineering

Faculty of Engineering

The University of New South Wales

Combining garbage collection and region allocation in DDC

by

Hui Yu

Thesis submitted as a requirement for the degree of

Bachelor of Engineering (Honours) in Software Engineering

Submitted: Oct 2018

Supervisor: Dr. Gabriele Keller

Co-Supervisor: Dr. Ben Lippmeier

Student ID: z5055838

Abstract

Traditionally, garbage collectors reduce that overhead by reducing the chance of handling long-lived objects. We present an approach by combining manual memory management with automatic memory management to reduce the performance impact of long-lived objects and show the preliminary test results of this method. The method allows the programmer to have more control of garbage collector behaviours while still maintain a clean and safe automatic memory management abstraction.

Thanks

I would like to thank my co-supervisor, Ben, for his sincere and valuable guidance extended to me, and banned me from CFD to prevent me from bankrupt.

I wish to express my sincere thanks to Gabriele Keller for all the courses taught and inspired me to learn functional programmings and programming language theory.

I would like to thank all the friends help me with reviewing my drafts, Alva, Chris, Jacqueline, Raymond, and Zheng.

Finally, I would like to thank my parents for supporting me and encouraging me over years.

Contents

Chapter 1

Introduction

1.1 Garbage Collection

Garbage collection is used in many modern programming languages, such as Java, C# and Haskell. Garbage Collector has an indispensable role in the language runtime system. Garbage collection ensures that the programmer does not need to worry about managing memory, relying on the runtime Garbage Collector to de-allocate objects when they are not used after a particular state of the application. The Garbage Collector eliminates a significant portion of bugs concerning manual memory management, and boosts the development efficiency since developers focus on real programming logic, rather than memory management.

1.2 Motivation and Goals

Garbage Collection has drawbacks. The garbage collector introduces runtime overhead, as it needs to pause the execution of the program or a least a thread of the program. The tracing garbage collector, for example, traces every reachable object in one cycle of garbage collection and then copies or compacts the live objects. The overall pause time varies from program to program. One program can have a short pause time, since

objects tend to have short lifetimes so that the cost of copying long-live object will be low. Another program might have a very long pause time since objects tend to be live for an extended period. The garbage collector does not know the exact lifetime of an object before program runtime. The programmer, on the other hand, knows that some objects will no longer be useful for the program at some specific time. However, in most current systems, it hides the entire memory management from programmers. Programmers do not have the direct control of how the garbage collector works when writing their code.

The primary purpose of this thesis is to implement a system inspired by region memory management so that the programmer can give hints to the garbage collector while coding to avoid unnecessary pause time on long-living objects, in order to improve overall throughput whilst retaining a safe and automatic memory management abstraction.

The secondary purpose is to improve the cache performance by grouping logically related objects together. For example, to improve the cache performance of iterating over collections by placing items of the collection into a contiguous memory space.

The development has been done in a research compiler, the Disco Discus Compiler, which compiles Discus language. The structure of Discus language will be introduced in the background section further on.

Chapter 2

Background

This chapter gives a brief overview on memory management techniques, particularly, garbage collection and region-based memory management. This chapter also introduces the background and runtime system of the Disco Discus Compiler and related works such as compact normal form and garbage collecting regions.

When a program executes, it stores the code, along with the data used for computation in the memory space. The Operating System(OS) provides an abstraction of memory space for a process. A programming language will need to utilize the OS' abstraction in order to provide a memory management abstraction on the language level.

2.1 Stack Discipline

The stack discipline is one of the memory abstraction mechanisms that the programming language may provide. The life cycle of objects is determined at the compile time, and no object can survive out of the local scope. Given a code block, the stack discipline explicitly determines the memory space that objects are allocated in the code block by moving the stack pointer. All local objects are freed automatically when control reaches the end of the code block.

However, stack discipline has its limitations. One problem with only using stacks is that they do not handle objects whose lifetimes do not follow the syntactic structure of the code. Another problem is that the data structures with unknown length, like dynamic lists or arrays, are not supported in the procedure call if using the stack-based memory management because stack space cannot be allocated if the length of the data structure is not known. [TT97]

2.2 Manual Memory Management

To solve the problems of stack discipline, programming languages like C wrap the system call to claim memory pages to a language built-in function *malloc*. It has a corresponding function *free* to release the malloced memory space. The data allocated this way is placed on the heap. C releases the control of object's lifetime in the memory space to the programmer so that programmer can manually allocate and then de-allocate the memory when the data is not used anymore.

The memory used by heap objects allocated by malloc can only be freed in two ways, the process exits or programmer calls free in the code. By having this property, dynamic arrays and lists can be allocated on the heap and then pass by a pointer to the address of the start of the list. Memory space can be shared between different computations if used properly.

The price for this freedom of control on the memory in the programming language level is extremely high. The programmer needs to think more about memory management while coding. To be more specific, the programmer might forget to call free function on allocated heap objects so that the memory space never is reclaimed (memory leak). When calling free function to free a heap object referred by two or more pointers, other pointer is still referring to a 'free' heap object. Such pointer is known as a *dangling pointer*. If further computations access the data pointed by the dangling pointer, the behaviour of the program will be unpredictable [JHM12].

2.3 Garbage Collection

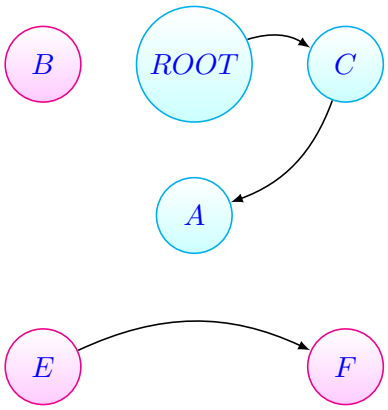
Instead of managing the memory manually, most of modern programming language runtime system have a *garbage collector* (known as *collector* or *worker*) to manage the heap memory of the application (known as a set of *mutators*).

2.3.1 Interface

The garbage collector provides an interface to claim memory space and this process is known as *allocation*. The reclaiming, known as *de-allocation*, is managed by the garbage collector automatically. The mutators utilize the interfaces to allocate *objects* on the heap. *Objects* have similar a concept in the object-oriented programming, that has *fields*. A field is either a *reference* to an object or a language specific *primitive object*. De-allocation is performed by a set of garbage collection cycle. A garbage collection cycle usually has two steps *identification* and *reclaiming*.

2.3.2 Identification

Identification is the process of identifying *live* or *dead* objects on the heap. Dead objects, known as unreachable objects, are referred as the garbage to be collected [Cha87]. An intuitive example to define the live and dead object is to use a directed graph. The graph below shows a memory heap state that has object named {A,B,C,D,E,F} and a ROOT which is the *root* object. Root objects are accessible outside of the heap, such as the variables in the stack that contain pointers into the heap, or global variables that contain pointers into the heap.



Live objects are objects that can be reached from the root by traversing the graph from root while dead objects are the objects that *unreachable* from root objects such as {B, E, F} in the graph.

The definition of liveness of an object may vary. In [JHM12] page 13, a live object is an object that at certain execution state, is guaranteed to be accessed in future executions of the mutator. On the same page of the book, it proves the liveness in this definition is undecidable for an arbitrary program and falls back to the weaker definition, which is reachability definition in the previous example with the graph. The author of the book gives an example, which proves that reachability is a weaker property than liveness. The fact that an object holds a reference to another object does not mean that mutator will access the referred object.

2.3.3 Reclaiming

Reclaiming is a process of freeing the memory used by dead objects identified in the identification stage. Dangling pointers are eliminated since the garbage collector only reclaims the object when it has no references from others, or in a sub-graph which is not referred by the root objects. Once the garbage collector has finished the reclaiming stage, the garbage collection cycle has completed.

2.3.4 Tricolour Abstraction

More formally, a tricolour abstraction describes the property in the identifying and reclaiming phases. The tricolour abstraction uses invariants to classify the objects in the garbage collection. In the beginning, all the objects are *white*. The identification phase comes through and introduces another colour — grey. When the garbage collector tracing process first encounters an object, it marks the object as a grey object. When the marking of that object is finished, and garbage collector starts marking its children, the object is coloured black. All black objects are live. The general use of this is to prove the correctness of the garbage collector such that an invariant persists: no reference exists from black objects to white objects. [**?**]

2.3.5 Evaluation

Many different garbage collection algorithms have been presented since the first one by John McCarthy in 1959 [Mcc60]. They all have improvements in term of performance or memory efficiency. To properly evaluate a garbage collection algorithms, following metrics can be used [SM06] :

- **Pause time**: the length of time during which application execution is stopped while garbage collection is occurring
- **Throughput**: the percentage of total time not spent in garbage collection, considered over long periods of time.
- **Garbage collection overhead**: the inverse of throughput, that is, the percentage of total time spent in garbage collection.
- **Frequency of collection**: how often collection occurs, relative to application execution.
- **Promptness**: the time between when an object becomes garbage and when the memory becomes available.

Different garbage collectors have different implementations of allocation, identification and reclaiming. Some garbage collectors always perform *stop-the-world* garbage collection, which means the mutators stop progressing within the garbage collection cycle. *Concurrent* garbage collector, on the other hand, allows the mutator to make progress in the garbage collection cycle. However, generally, a pause on mutator is inevitable [SM06] given that memory space is not infinite.

2.3.6 Garbage Collector Implementations

This subsection introduces various garbage collection algorithms and the drawbacks of the pause time. One may find implementation details of these algorithms in [JHM12].

Mark-Sweep

The mark-sweep garbage collector [Mcc60] (MSGC) is a tracing garbage collector. It has two phases, the mark phase and the sweep phase. The mark phase starts tracing from *root* objects. The garbage collector marks the reachable objects as live objects. Then it starts a sweep phase. During the sweep phase, the garbage collector traverses the heap and reclaims memory which is not marked by the garbage collector and resets mark if the object is marked as being live.

The mark and sweep garbage collector has its drawback of fragmentation. The fragmentation leads to a situation where there are many slots available, and the total available size of memory is enormous, but none of the slots is large enough to allocate the object. The nature of sweep causes it, it creates holes between objects and separates the heap into small space fragments [JHM12].

Mark-Compact

The Mark and compact garbage collector [HW67] (MCGC) targets the problem of fragmentation present in the mark-sweep garbage collector. The Mark-compact garbage

collector compacts all the live objects into a chunk of memory by moving objects into the gap while leaving the free space in one piece after each garbage collection cycle.

By compacting the objects into a chunk a memory space, it boosts the cache performance. However, the algorithms to achieve compaction usually requires multiple passes over live objects, and in an overhead that can be avoided [Cha87].

Copying

Copying garbage collection usually refers to the semispace copying collection proposed originally by Cheney, C. J. The garbage collector divides the heap into two spaces, *from-space* and *to-space*. The free pointer points to the start of free space within the from-space. Objects are allocated into the from-space by *bumping* the free pointer to achieve fast allocation. A garbage collection cycle starts when the from space is full. All the root objects are first copied to the to-space and then start identification phase by tracing from root objects. In the identification phase, root objects are copied into the to-space first. A new pointer *scan* is introduced. The scan and free pointers point to the end of last black objects. During the trace, the grey objects are copied into the to-space and then bumps the free pointer to the end of grey objects. The garbage collector scans through objects by bumping the scan pointer, and copies objects traced in the from-space to the to-space, and then bumps the free pointer. Repeating the copy sequence until scan pointer has the same address of the free pointer, the algorithm terminates. The reclaiming phase is simple, swap the role of the to-space and from-space [Che70].

The benefit of this elegant algorithm is that it reduces the costs of compaction significantly. However, it introduces the cost of copying. Large objects which survive for a long time are copied over and over which requires time. Another drawback is the available heap space is only half size, compared with other garbage collectors.

Generational Garbage Collector

The performance of the tracing garbage collector is dependent on the number of live objects. As the number of live objects decreases, the faster the garbage collection cycle will generally be. One way to achieve this is to partition the heap into different memory spaces. Garbage collector only runs on one space at a time so that the mutator could access objects on other spaces. However, this model has an issue: objects loosely distribute over all spaces. One garbage collection might need to clean multiple spaces. A policy to arrange objects is needed. One garbage collector solves the problem effectively, based on the generation scavenging algorithm [Ung84]. It is called the generational garbage collector (GGC). The GGC is based on the generational hypothesis that says most objects die quickly and the old objects are expected to have higher survival rate by observation [Hay91]. The heap partitions into two or more independent spaces called *generations* by the amount of GC cycles an object *survives*(not marked as dead object). The GGC can perform garbage collection on each generation separately so that the garbage collector working on generation A does not block the mutator to access a generation B.

The youngest generation, where the garbage collector allocates the object to, is called the Eden or Nursery. Most objects allocated die here shortly. From the study of SPECjvm98 benchmark suite, only 9% of objects survived after a 4MB allocation [DH99]. Most of the garbage collection cycles happen here. The GGC perform minor garbage collection on the nursery and then *promotes* the surviving objects into older generations so on and so forth.

To *complete* the minor garbage collection, extra information of an object must be provided from outside of the heap. An object in the nursery can be kept alive by having a reference from an older generation. The remembered set is introduced to achieve this without tracing other generations. The *write* operation specifies that when an inter-generational reference creates, it records the source of reference into the remembered set. In the garbage collection cycle, objects in the remembered set are added to the root, if the object is still live.

The modern generational garbage collector has multiple generations, extra layers to give a buffer to old generations in order to reduce pause time garbage collection within old generations. In term of the pause time, the generational garbage collector has improvements, yet still fail to resolve the issue of long pause time when collecting the old generations.

Reference Counting

The Reference counting garbage collector is published in [Col60]. In the paper, it describes that each object maintains a counter when allocated by the runtime system. The counter increases when a new reference is pointed to the object, and decreases when a reference is removed. The nature of the reference counting garbage collector needs additional interface the runtime system. To avoid concurrency issues on the mutator side, a *write* operation is an *atomic* operation provided by the garbage collector. When updating a field in an object, the *write* is called to decrease the counter of old objects, then increases the counter of the new object the field is going to refer.

The reference counting garbage collector makes the identification on the fly. When the counter of an object drops to zero, it is classified as garbage. One can observe the further effect of this, the counter of other objects to which the dead object referred decrements as soon as the dead object marked as garbage. The time overhead is imposed on the mutator to update the counter in the object header.

We assumed the *write* operation is atomic; yet automicity comes at a price. In a racing condition, two mutators attempt to increase the counter of an object concurrently. One has to wait for another to finish. In the worst case, all the mutators attempt to increase the counter of an object and the pause time of the last mutator which increases the counter is high. Moreover, when an object counter drops to zero, reference counting garbage collector must update the counts of all the fields of the object. If the object has a large data structure, all the references in the fields of the object will need to be decreased, as well as their fields. The stop-the-world deallocation, in this case, will pause the mutators for a long time.

The Reference counting garbage collector must perform extra graph iterations to finish the garbage collection for cyclic data [BCR04]. To identify the cycle in the heap, cyclic reference counting algorithm like the *Recycler* [BR01] uses a concurrent reference counting algorithm reduce the pause time on the mutators, however, it still takes $O(N+E)$ time where N is the number of node in the graph (objects) and E is the number of edges among nodes (references).

2.4 Region Based Memory Management

2.4.1 Effect and Region

The idea of *Region* type was used in *effect system* [LG88] where work has been done over last three decades. In the book [TGS08] on page 944, region is defined as an abstract box to store the *single* object and each region can be considered as a distinct memory block, or a network URL where information can be stored. Operations with an *effect* such as *Read*, *Write* and *Alloc* can be performed on the region.

2.4.2 Region Inference

In 1992, a paper [TJ92] describes a method to automatically manage memory by using *region inference*. Region handle can be put into a stack and every expression in a well-typed language (*source language*), which can be translated into a *target* language that has region-annotated expressions describing the life cycle of regions (i.e. from allocation to de-allocation). Two forms of region annotated expressions are:

- e_1 in ρ
- letregion ρ in e_2

For the former, the *storable value* computed by the expression e_1 is in the region bound to the region variable ρ. The latter one introduces a region, bound to region variable

ρ with local scope e_2, and when the call finishes, the objects inside the region are deallocated. This makes the regions allocation and de-allocation like stack-manner.

Talpin specified the region inference rules for a *pure* language in the paper. Generally, the inference rules allow an express e to translates into e' which has the type τ and region ρ and the effect ψ in the type environment TE

$$TE \vdash e \Rightarrow e^{'} : (\tau, \rho), \psi$$

The problem with region-based memory management is that it requires the program to be written in a region-friendly style for the underlying region inference algorithms. The program will leak memory if the programmer fails to do so. In the worst case, the entire heap is live even most of the regions are safe to be de-allocated. The details will be discussed further in section 2.4.5.

2.4.3 Region polymorphism

Two problems of the model mentioned previously and solutions with proofs were mentioned in the paper publishsed later [TT97].

- A function in source language translated by the region inference algorithm will need to keep the return region alive until all function calls have returned.
- In term of recursion, the result of the recursive call will have to be in one region.

To solve the first one, functions are translated in such way that a region variable can be taken as a parameter of the function. Extra annotation for the function translations are needed to access and make the region polymorphic function possible:

- letrec $f[\rho_1, \rho_2 ... \rho_n](x) = e_1$ in e_2
- $f[\rho_1^{'}, \rho_2^{'} ... \rho_n^{'}]$

For the second problem, Birkedal and Tofte came up with an algorithm called Birkedal storage mode analysis. The algorithm is designed to reset the regions which are not used in the further recursive call and *overwrites* the region.

2.4.4 Finite and Infinite Regions

ML Kit is a compiler for Standard ML code and it is the first large-scale compiler that uses region inference. The performance of program compiled by ML Kit was poor - it was slow, and used smore memory than the state-of-art compiler ML of New Jersey. During the testing and development of the ML Kit, Birkedal, Tofte and Vejlstrup found that most of the regions contained only one value. Based on this observation, they conclude that a *finite* region — that is the region with fixed length — can be optimised to store in the activation records, while an *infinite region* has to be stored into data structure like a linked list and each of the node is a fixed-size *region pages*. When a linked list for the region is full, free region pages are allocated from a list of free region pages. The method is called *multiplicity inference analysis*.

The performance is roughly faster by a factor of three for the finite and infinite region implementation [BTV96].

2.4.5 Garbage-Collect Regions

In the early 21st century, Hallenberg in his paper [HET02] describes a scheme to use a modified semi-heap garbage collector to collect regions. The implementation is in the ML Kit, and fully tested evaluations of the method are given by comparing it with performance of pure region discipline or pure garbage collection.

Each region is associated with a *from-space* and a *to-space*. During the garbage collection, Cheney's algorithm is applied to each local region to perform a copy garbage collection. The root objects are the *region descriptors* in the *region stack*. To keep trace of region pages getting garbage collected, all the pages in the from-space are

linked and form a large global from-space. The garbage collector identifies live regions by traversing from each region stack and finds the region that it points to. In each traversed region, the garbage collector copies each page in the region to a new page from the free-list. When the garbage copies are finished, all the region pages in the from-space will be appended to the free-list.

The benchmark shows that if the program optimises for the region, the garbage collection with region inference has an unfavourable effect than if region inference is used only, in term of performance and memory consumption. The reason behind this is the pause time of garbage collection and the extra space used by the to-space. However, the programs that have not been optimised for regions inference, all use less memory space when it runs with garbage collection than using region inference alone.

In term of the pause time, the region garbage collection has the same issue as the semi-space garbage collector, since every object in the from-region space eventually will be scanned and copied to a free region in the free list.

2.4.6 Compact Normal Form

The other related work is the compact normal form (CNF) introduced in [YCA+15] to reduce the overhead of serialisation while transferring data over a distributed system in Haskell. A CNF is an abstraction of contiguous memory space with restrictions. The objects in a heap can be copied into a CNF (*allocation*) and then the objects in CNF are 'compact' and 'self-contained'.

Three properties are defined for the CNF:

- No outgoing pointers: The Objects in the CNF are only allowed to have reference to the objects in the same CNF.
- Immutability: The mutability is not allowed for objects in the CNF.
- No Thunks: The function closures are not stored in the CNF.

The first property has benefit in the garbage collection. If an object is completely self-contained and in a contiguous space, then no further tracing will be needed to keep its children live. Long-lived data can be placed in a compact region to exclude them from garbage collection so that the pause time of a garbage collection cycle is reduced. [YCA+15]. The CNF will be discussed in details in later chapter.

2.5 Discus

The *Discus* (was called *Disciple* [Lip10]) is a strict dialect programming language of Haskell. The language supports *higher-rank polymorphism*. It also has modal region and effect system in the language. The Disco Discus Compiler (DDC) is the compiler for the Discus.

2.5.1 Region In Discus

Region types are widely used in Discus. A region has *capability* to handle effects of operations on the data stored in the region. For example, any function that updates a reference(*Ref*) may have effect type *Read* and *Write* which is on the region the (*Ref*) is stored in.

The built-in annotation allows the introduction of a region, or the extension of a region. The use *private* primitives allow the introduction of a new local region by specifying the region name and the effects allowed on the region. Use of the *extend* constructor allow the extension the existing region with the new subregion. The extension requires the name of *subregion* and effects allowed on the *region*.

This makes the destructive update syntax looks neat. For example, the Ref object is a container of a pointer to another object. The interface functions for Ref have types:

- allocRef :: $\forall r$: Region. $\forall a$: Data. $\rightarrow$ a $\rightarrow$ S (Alloc r) (Ref r a)
- readRef :: $\forall r$: Region. $\forall a$: Data. $\rightarrow$ Ref r a $\rightarrow$ a $\rightarrow$ S (Read r) a

- writeRef :: $\forall r$: Region. $\forall a$: Data. $\rightarrow$ Ref r a $\rightarrow$ a $\rightarrow$ S (Write r) Unit

The effect of the function is described by side effect constructor **S** alone with the return value of the function. The syntax is elegant in that the effect information is in the code and the effect is more specific compared with *Monad* in Haskell.

2.5.2 The Disco Discus Compiler

The Disco Discus compiler compiles Discus code into binary code by going through different compile stages. From the compiler perspective, a *source* language is the language that the programmer writes. It has syntactic sugar, and the program is not explicitly typed. The DDC parses the source language and performs type checks, type inferences and de-sugar the syntactic sugar, which then produces a *core* language. The *core* language is based on System F, which formalises parametric polymorphism. The *core* language has two sets of primitives. One is for the *Salt* which is the language to write the DDC runtime system. Another set of primitives are called the *Core Discus* primitives, which is just the primitives corresponding to the source language. Core language and runtime written in Salt or C are then compiled together into LLVM, and then, compiled by LLVM to object code.

2.5.3 DDC Runtime System

The DDC Runtime system is mainly written in *Salt*, which has the concrete syntax of *Core language* but has its primitives to access memory on the heap for example. The *Salt* could call C functions, but it is slower than 'function' in Salt because Salt function call can be inlined. It is significantly faster than C function calls that build stack frames and return values.

The runtime system has a few types of objects (use 64-bit objects as an example) [Lan17].

- **Thunk**: a partially applied function closure.
- **Boxed**: an object with pointers to up to $2^{24} - 1$ other boxed objects.
- **Array**: an object with pointers to up to $2^{32} - 1$ other boxed objects.
- **Raw**: an object containing up to $(2^{32} - 1 - 8)$ raw non-pointer bytes.
- **Small**: an object containing up to 4 raw non-pointer bytes.

Some region information is exposed to the runtime system because of Ma's previous work [Ma12]. The goal of Ma's work is to provide the information about regions in Discus to the LLVM to optimise the LLVM compiler code generation, by specifying Type Based Anti Aliasing (TBAA) metadata.

The runtime system has a copy garbage collector, which is the semi-space garbage collector model mentioned in section 2.3.6.

Chapter 3

Compact Region

This chapter introduces the compact region and its properties. Firstly, We will explain the problem that we are solving and motivation. Then we define the properties of a compact region.

3.1 Compact Region

In section 2.3, we described many garbage collector implementations. These garbage collectors have various optimisations to reduce overhead on the long-lived objects, however, at some execution state, a garbage collector eventually has to deal with the long-lived object. For example, the generational garbage collector eventually copies relatively long-lived objects when a promotion is required. We argue that the long-lived objects are one of the reasons garbage collector takes more CPU cycles than it needs.

An obvious approach is to ignore these long-lived objects, and only garbage collects them when they are dead. However, from early section 2.3.2, we know the lifetime of an object is unknown until the object is reclaimed. Within a current fully-automated memory management model, this approach would never work.

We assume that programmers tend to know when the object is dead most of the time

when they create a long-lived object. For example, a game developer knows that the map object lives until the player goes to another map. We would like to take advantage of the observation and give the programmer a degree of freedom to manage the memory so that they could decide when the object is dead at a particular state.

We now introduce the compact region. A programmer can put the long-lived objects into the compact region and then set the lifetime of the compact region. The garbage collector now has the missing information and assumes that objects in the compact region stay alive before the compact region is dead.

The idea of the compact region is originally from the Compact Normal Form (CNF) [YCA+15]. The CNF is defined as a self-contained region, where each object in the region can only have reference to other objects in the region. We consider this as the first step of the definition of the compact region — a collection of objects that can be logically isolated from others. An object that is isolated from the objects in the heap means that the garbage collector does not manage the object, so that the tracing and copying processes will not happen for these objects. If a runtime system has compact region implementation, a valid object exists either in the compact region or in the heap.

The compact region has restricted interface compared with the well-known collection abstraction since it only allows for the insertion operation. Once the object is in the compact region, it cannot be deleted from the compact region before the entire region is marked dead. In our definition, we defined that the objects in the compact region tend to have a similar lifetime, and tend to have a longer lifetime compared with the object in the heap. However, the lifetime of the compact region is not decidable by the garbage collector — it is the programmer who defines this. Ideally, the objects have the same life in the compact region so that when the programmer uses a command to indicate the region is dead, all objects in the compact region die at the same time. Moreover, we would like to see that the object in the compact region is not referenced after the compact region is dead. However, it is impossible to determine the exact lifetime of the object when the programmer stores it into the compact region, and it's rare to have objects that die at the same time with no incoming references. Extra work

needs to be done to handle all the scenarios and all the cases, as well as the solution, will be introduced in this chapter.

3.1.1 User Interface

Programmer utilises the compact region through interface functions. We defined the following functions for the compact region:

1. **allocCR :: Unit → CR**
2. **storeCR ::** $\forall a.$ **CR → a → a**
3. **dissolveCR :: CR → ()**

The first interface function is the allocation of the compact region. It allocate a compact region and returns a region handle has type CR. The region handle is the unique identifier of the compact region.

The second interface function is the function to store an object into the compact region. The function takes two arguments, the first one is the region handle, and the second one is the heap object that the user wants to store into the compact region. It returns an object that is **in** the compact region. When an object is **in** the compact region, the lifetime of the object is free from garbage collection logic, and it is bound to the life of the compact region. In another word, the garbage collector never touches the object in the compact region until the region is dissolved.

The third interface function is the function to dissolve the compact region. It takes the compact region handle and the compact region is logically "dead". When a compact region dies, the object in the compact region is "out" of compact region, and considered as a normal object in the heap. To avoid dangling pointers, objects that are "out" of the compact region may not be reclaimed directly.

3.1.2 Garbage Collector Behaviour

The compact region is more complicated in practice. In this section, we discuss the various scenarios that could happen when using the garbage collector with the compact region and the garbage collector behaviours when garbage collecting the compact region.

A pure compact region implies a compact region that is self-contained which means it does not contain an outgoing pointer from the compact region to the heap or other compact regions. It means the compact region only contains raw binary data or the object only has reference to objects in the compact region. An example of that is a compact region that contains the entire texture data for an FPS game. The pure compact region is the simplest form of the compact region — however, to garbage collect the pure compact region and achieve the goal of elimination of the copying and tracing the long-lived object, the garbage collector for a standard runtime system needs to add an extra phase to garbage collect the compact region when they are dead. One possible implementation is to have a global list of the pointer which refers to the compact region, and the garbage collector walks through each pointer to find out dead compact regions. This phase happens at the end of tracing or copying objects for the standard garbage collector.

Beside that phase, when the garbage collector traces an object that is located in compact region, it needs to stop tracing immediately. If the garbage collector traces an object into a dissolved region, it runs the garbage collection logic like it garbage collects the heap object. As garbage collector automatically ensures no-dangling pointer, in this case, we could take the advantages of the garbage collector and ensures objects in a dissolved compact region, which still being referenced by an object in the heap, to copy them back onto the heap.

However, if the compact region is not pure anymore, which means it has outgoing references onto the heap or other compact regions, the garbage collector described above will cause dangling-pointer. Consider the case that a Ref object, which contains a pointer to other objects. If it contains a pointer that points to a natural number in

the heap. When doing the garbage collection, the natural number will be labelled as dead because it did not trace into the Ref object in the compact region and the garbage collector never reaches to the natural number. In this case, to solve this problem, we introduce a reference list to the compact region. Each compact region has its reference list, which contains the possible outgoing pointers from objects in the compact region. When an object is stored into the compact region, the garbage collector scans the fields of that object and add the pointer to that object to the reference list if an out-going pointer presents. When the garbage collector collects the heap, it needs to scan the reference list of a compact region and trace object in that list.

Some languages include the destructive update - the object's field can change once it has been allocated. In this case, the garbage collector described above won't work. For example, If the pointer in the Ref object which references to natural number one has changed to refer to a natural number 2, the reference list in the compact region header does not update itself so that in the next garbage collection cycle, there is a dangling-pointer in the filed of Ref object. We have various solutions to this problem. The simplest solution is to instead of keeping a reference to the objects in the heap in the reference list. It keeps the reference to the object in the compact region. When garbage collecting, it traces into the object in the compact region if there exists a pointer in the reference list. By doing so, it creates extra overhead that if the object has reference to other objects in the compact region, it still needs to trace one step down. Moreover, it may need to add a pointer for every object which has capability of the destructive update to the reference list, which creates enormous overhead.

A better method we introduce is to apply the proxy design pattern for the objects stored into compact region, and its field can potentially change. The proxy object adds additional functionalities so that when the field changes by a setter function, it checks if the new refered object is in the heap. If it's in the heap, then add it to the reference list and delete the old one from the reference list. It requires that the reference list data structure supports deletion.

Chapter 4

Implementation

In this chapter we present the implementation of compact regions in the Disco Discus Compiler (DDC) with discussion of problems and critical issues we encountered.

4.1 The Disco Discus Compiler

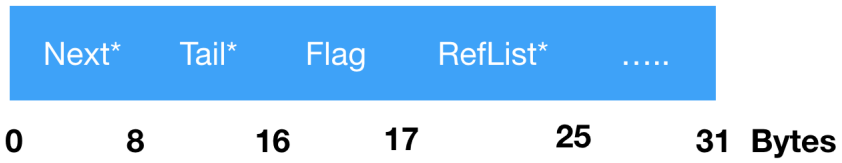
4.1.1 Salt Language

DDC compiles the source language Discus into Core languages and then compile into machine language by LLVM. The compact region implementation is one of the libraries of the DDC runtime. It provides interface functions to the source language, and most of the code is in the Salt language. Salt is a fragment of the DDC Core language for the runtime system. According to Ben Lippmeier, Salt is what we got when we left C out in the sun for too long [Lip10]. The compact region implementation is mainly in the DDC runtime system using the Salt language. Salt language has primitives like writes and reads an address.

4.1.2 Compact Region Data Structure

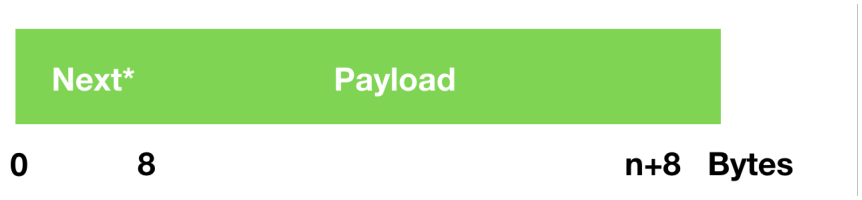
The compact region implementation in DDC uses a chunk of memory for each compact region. Initially, we were using the contiguous chunk of memory for each compact region. However, we found an issue that is not solvable in the current set up. Pointers may not be updated properly if we were using contiguous chunk of memory when the compact region extends. In the contiguous chunk memory implementation, the extend function will be called when it attempts to store an object into a compact region having insufficient space left. The extend function will claim a larger chunk of memory and copy all object in the compact region to the new one. In this case, any pointer points to object in the compact region immediately becomes to a dangling pointer. The runtime system cannot solve this without starting tracing objects and updates corresponding pointers when the extend function is called.

Instead, we use the linked list for the underlying data structure of the compact region. The header nodes have 25 bytes of memory. It has the 8 bytes pointer to the next node and 8 bytes pointer to the tail node and a byte for flags and 8 bytes for the pointer to the reference list. The first bit of the flag fields indicates if the compact region is live.



Compact Region Header

The node, which is in the compact region linked list but not the head of the list, has a simple header with only one field, the next field. The size of a node in total is 8 + n, where n is the size of the payload.



Compact Region Node

In the Salt language, we do not have the built-in feature to define a data structure. We use getter and setters function to access the memory. The payload is the object stored in the compact region. For the reference list, it has the same structure as the compact region, however, the reference list field in the header is always zero.

4.1.3 Pure Compact Region

Recall that the pure compact region stores objects that have no outgoing pointers. We start from introducing the implementation of pure compact region and the garbage collector modifications and then introduce the reference list implementation.

Allocation In Discus, each side effect is associated with a static region variable. A static region is like a container for data and it has its ability to read or write the data. The compact region is a physical chunk of memory. Allocation has the side effect of claiming a new chunk of memory. The interface function in Discus source language needs to take a static region, which is for the reasoning on the side effect, and return a handle to a physical compact region. The allocation function has type:

- **allocCR :: {@r : Region} → S (Alloc r) (CR r)**

It means the function allocCR takes an arbitrary region **r**, and return a compact region associated with the static region **r**.

When the interface function is called, it allocates the memory by calling the **malloc** function. It will only allocate the compact region head and set the next pointer and

tail pointer to itself. The first bit of the flag is set to one to indicate the compact region is live. To keep track of the compact region, a pointer to the compact region head is added to a global list called compact region table (CRT). CRT a stores compact region allocated and the garbage collector will access the table in each garbage collection cycle to garbage collect the compact region. CRT has a default size of 1024 in the current implementation, and it can extend itself by claiming a new chunk of memory, and then copy existing members of the list to the new chunk of memory and update the global pointer to the head of the CRT. It may need synchronisation if the runtime system has a concurrent garbage collector. However, DDC runtime has one thread and one process, we did not use any synchronisation techniques in this matter. The region handle is just the index of the CRT and we utilise the DDC type system to make sure that fake compact region handles cannot be constructed.

Storing Objects We intend to make the compact region polymorphic so that values with different types can be stored in a single compact region. In the runtime system, we have small, raw, boxed, and thunk object types to handle. The function type signature is:

- **storeCR :: {@r : Region} → {@a: Data} → CR r → a → S (Write r) a**

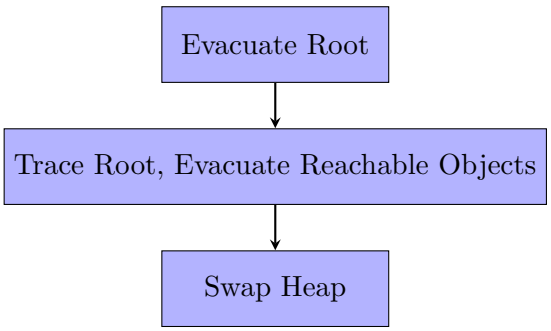
The function is parametrically polymorphic. It takes the static region of the physical compact region and the type of object storing into the compact region, and returns an object typed a. The function writes into the compact region, so the effect is described in return type using the effect constructer S and with a write effect on the region r. In the runtime system, recall that we use the linked list implementation for the compact region. A new node is created to append to the tail of the compact region linked list, and then the object is copied to the newly created node.

Dissolve Compact Region The last interface function is the dissolve function. The function has type signature:

- **dissolveCR :: {@r : Region} → CR r → S (Write r) Unit**

In the runtime system, we delay the physical destruction of the compact region to the next garbage collection cycle, since a reference from the heap object to an object in the compact region may exist. We accomplish it by setting the compact region live flag to zero so that the garbage collector knows the compact region dead.

Garbage Collector DDC runtime system uses the semi-heap copy garbage collector. It has a front heap and a back heap. As mentioned in chapter 2, the garbage collector allocates into the first heap and trace and copy live objects to the back heap then switch two heaps. The DDC garbage collector starts the garbage collection cycle by copying root objects and then trace the root objects one by one and copy reachable objects into the back heap.

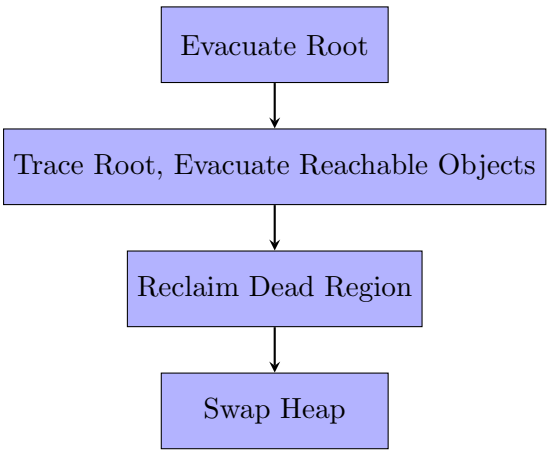


Garbage Collector Phases

Our goal is to reduce the tracing overhead of the long-lived object. The first modification to the garbage collector is to make the garbage collector stop tracing into the compact region we allocated. When tracing root objects, if the garbage collector traces an object that is in the compact region, the garbage collector shall skip the object and trace the next field of the root object. One possible implementation is to look up the compact region and see if the object exists in the compact region. It introduces a significant overhead, which even slows the garbage collection cycle. We make the time complexity to a constant time by using a flag in the object header. DDC runtime

object header has a flag field to distinguish the format of different objects. We found we could use a flag to indicate if the object is in the compact region and set the flag to true when storing it into the compact region and to false when programmer marks the compact region dead. The garbage collector reads the flag so that it takes constant time to decide if it needs to trace the object. We also utilise the garbage collector so that when the compact region is dead, the reachable object in the compact region shall be copied back to the heap. However, by doing so the dissolveCR function will have time complexity proportional to the number of objects in the compact region.

The second modification to the garbage collector is to add another phase after the evacuating the reachable objects from the heap. The garbage collector scans compact region in the compact region table and looks for the dead compact regions. It is safe to reclaim the compact region since all reachable object has already been copied to the heap. The time complexity is linear, which is proportional to the number of compact regions. In reality, we expect the number of compact regions is negligible compared with the objects allocated on the heap or in the compact region so that it does not introduce significant overhead.



Garbage Collector Phases (pure compact region)

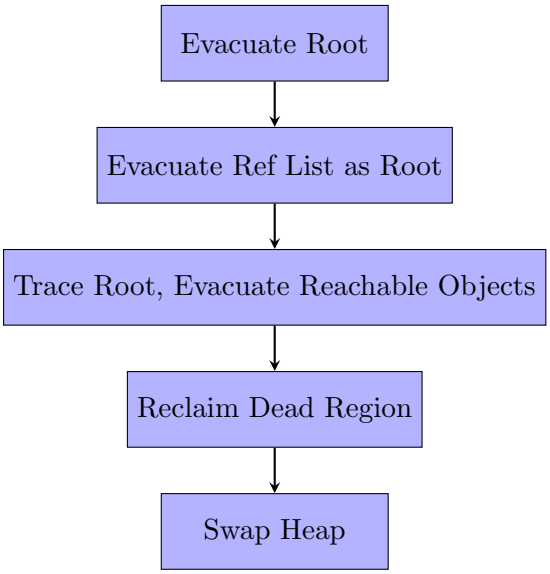
4.1.4 Compact Region With Outgoing Reference

The self-contained, compact region sometimes is not useful. A scenario is when the destructive updates are allowed, objects in the compact region may have outgoing references which points to the object on the heap. A simple example is an array of natural numbers, where the natural numbers are on the heap, and the object represents array is in the compact region. To make this work, we implemented the reference list mentioned in chapter three and supported the destructive updates since DDC supports datatype such as Ref.

We firstly change the compact region head. The head now needs to contain a pointer to the reference list. We reuse the implementation of a compact region so that the reference list itself is a compact region. The way to distinguish a real compact region and the reference list is the reference list does not have a reference list, i.e. that field is always 0x0. After storing an object into the compact region, if the object potentially has a field containing the pointer, we store a pointer to that object in the compact region to the reference list.

For the garbage collector, the first modification is to allow the garbage collector to trace into the references list, hence traces the fields of objects containing pointers in the compact region. Recall the DDC runtime garbage collection steps, we introduce a new step just after the evacuation of root objects. The garbage collector goes through the compact region table and finds the compact region which is live and then take the reference list of the compact region. The garbage collector continues to trace the objects pointed by the pointers in the reference list, not recursively, but only one step. The garbage collector copies the traced object if the object is on the heap and ignore it if it is in the compact region. An example of that is when a text object. In DDC runtime, a text object is composed of a boxed object which contains a pointer and a raw binary object which is the data. If the boxed object is stored in the compact region, and the raw object is on the heap. The garbage collector traces the pointer in the reference list which finds the boxed object on the compact region. The garbage collector then traces one more step, to reach the raw object and then evacuate the raw

object into the back heap. When this phase ends, the garbage collector starts tracing object on the back heap and evacuate reachable objects. We also take advantage of the garbage collector that the object copied in this phase trades as the root objects and the garbage collector will continue to trace objects from root objects in the next phase.



Garbage Collector Phases (outgoing pointer)

Chapter 5

Evaluation

This chapter presents the preliminary results of using the compact region to reduce the garbage collector overhead in the DDC. We use some example to simulate real-life scenarios when using compact region.

5.1 ToolBox

5.1.1 Statistics Module

We implement a statistics module to collect the information of the runtime system. It includes two parts. The first part is written in Salt, and it records values such as the total bytes allocated. The first part will print out the total bytes allocated and the size of the front and the back heap when the program exits. The second part is implemented in C. It keeps the pause time of the garbage collection cycles in a linked list and keeps appending to the linked list in each garbage collection cycle. It uses the clock in the time library, which is one of the C standard libraries, to calculate the number of CPU ticks spent on the garbage collection cycle and then convert it into seconds by using the constant CLOCKS_PER_SEC. We convert the result to the milliseconds. The second part of the module will print out the pause time of each garbage collection cycles stored

in the linked list when the program exits. One may use runtime argument enableStats to enable the statistics module when running a DDC program.

5.1.2 Test Program

We use a pattern to write out test program. The test program has two phases, the allocation phase and the computation phase. In the first allocation phase, the program will allocate data needed for the testing. The garbage collection cycles happen in this phase does not take account into our test results because allocation may trigger garbage collection and allocation function may cause slow garbage collection. An example to justify this is to allocate a list with over 60K elements. The allocation function will run recursively so that the heap will be filled with thunks. In the second phase, we run the exponential time complexity function to compute the 40th Fibonacci number. The function will create a lot of thunks and some of them will become disposable in a short period of time. It is used to simulate a real program in the production environment. For example, the web server may allocate a buffer and release it in a short period. It is one of the further work to test the compact region on a program that uses in the production environment.

5.2 Pure Compact Region

We implement a special function in the runtime system and import it to the Discus language. The function takes a natural number, which is the size of the object we want and allocates a raw object on the heap. DDC runtime system recognises object size by looking up a number in a field of the header. The function will set that field to 'fake' a raw object with payload while having nothing in its payload. The runtime system will reserve the memory space for the object. By using this function, we can get raw object for any size.

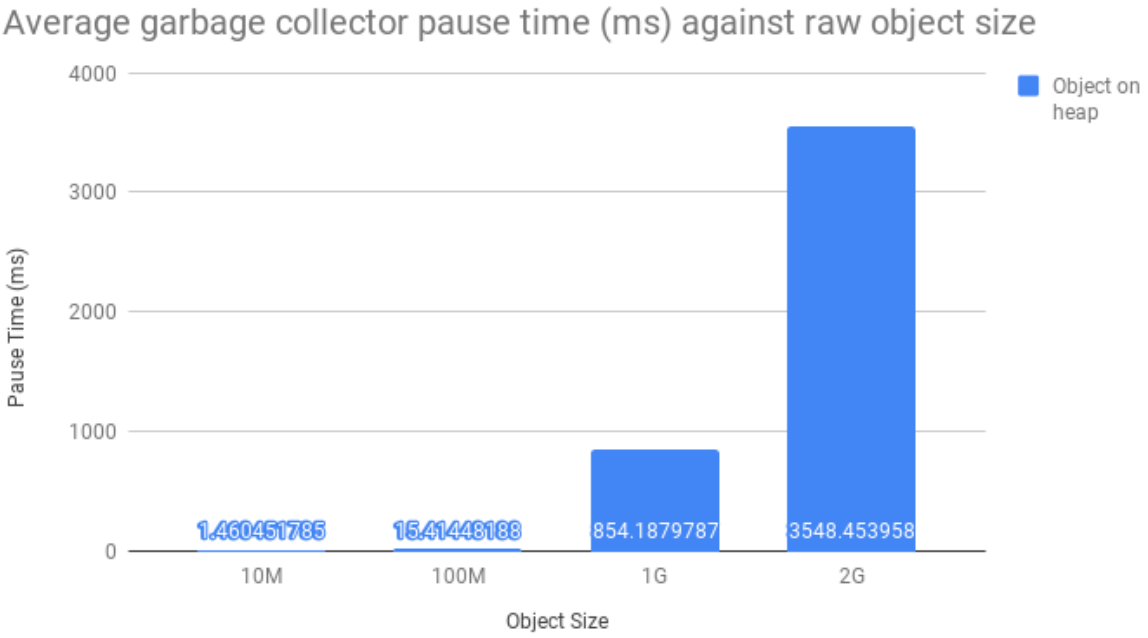
The first test group is to understand pause time of DDC garbage collector because

of long-lived objects. We allocate a raw object in the heap and start calculating the 40th Fibonacci number. The garbage collector will start running when computing the Fibonacci number and the raw object will be copied in each garbage collection cycle. We make the size of the raw object as an independent variable so that we can study the impact of the size of the long-lived object. The raw object size we used is 10MB, 100MB, 1GB and 2GB.

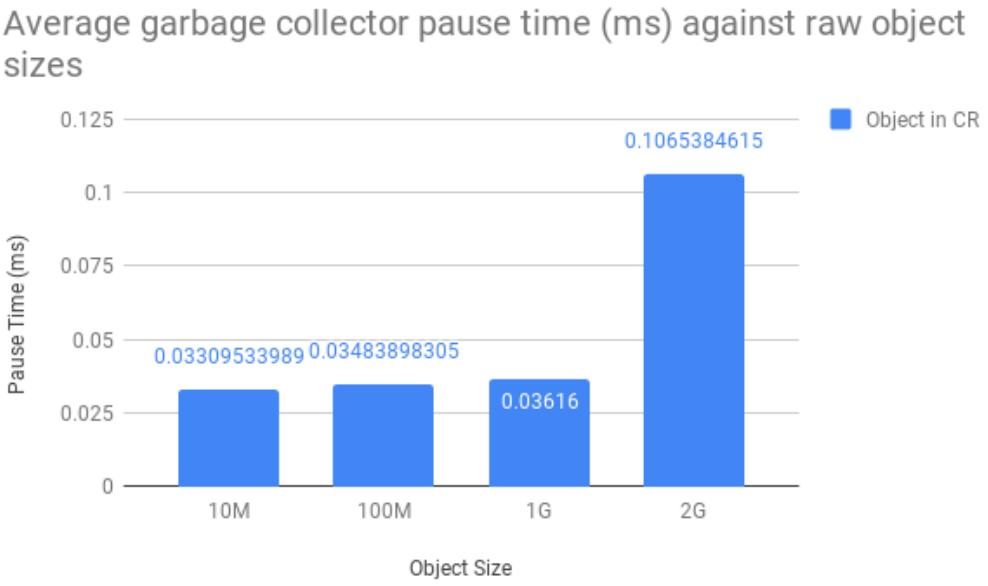
The second test group is to understand how pure compact region may reduce the overhead of the long-lived objects. Like the first one, we compute the 40th Fibonacci number. However, we allocate raw objects into a compact region. We also control the size of the raw object to study if the size of the object is relevant when using a compact region as the container. We use the same object sizes for this test group so that we could compare with the first group to understand the improvements by the pure compact region. We use raw object because of boxed object in a pure compact region will have reference to object in the compact region itself which is not going to be traced.

For both test groups, we ignore the pause time of the garbage collection cycles before computing the Fibonacci number to reduce the error may be caused by the allocation. We run all tests ten times and get the average pause time of the garbage collection cycle and compare results vertically and horizontally.

First test group For the first test group, we calculate the average garbage collection time while computing the Fibonacci number. The allocated object is already in the heap so that we expect the larger the object is, the longer it may take for a garbage collection cycle. From the graph, we could observe such relation. The pause time was 1.46 ms when the raw object is only 10MB and 3548.45 ms when it is 2GB.



Second test group For the second test group, the object is in a compact region so that it reduces time to copy the objects in each garbage collection cycle. We expect the pause time of the garbage collector to be negligible. We could observe the result is promising. The pause time for all object sizes is less comparing with the first test group.



We also observe the performance boost is relevant to the size of the object allocated into the compact region. For example, the 10MB raw object, it is 44 times faster when using compact region. For the 2GB raw object , it is 33000 times faster when using the compact region.

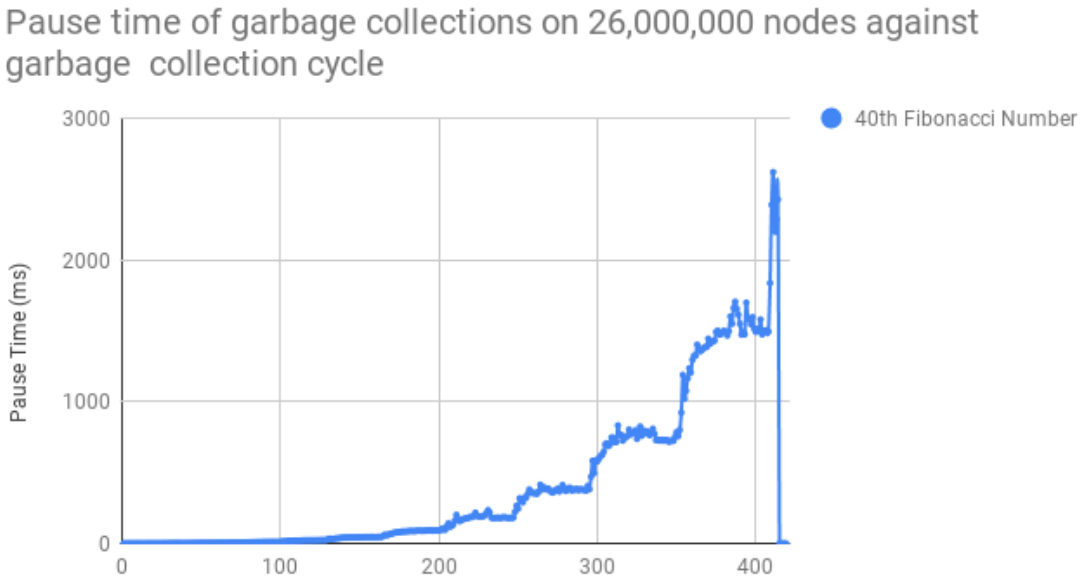
5.3 Compact Region And Outgoing reference

The pure compact region is only useful to store the object having no outgoing reference. It is reasonable that it has a significant improvement in the pause time of a garbage collection cycle. We hypothesis the outgoing references may increase the average pause time because of the extra trace and copy time we introduced by using the reference list.

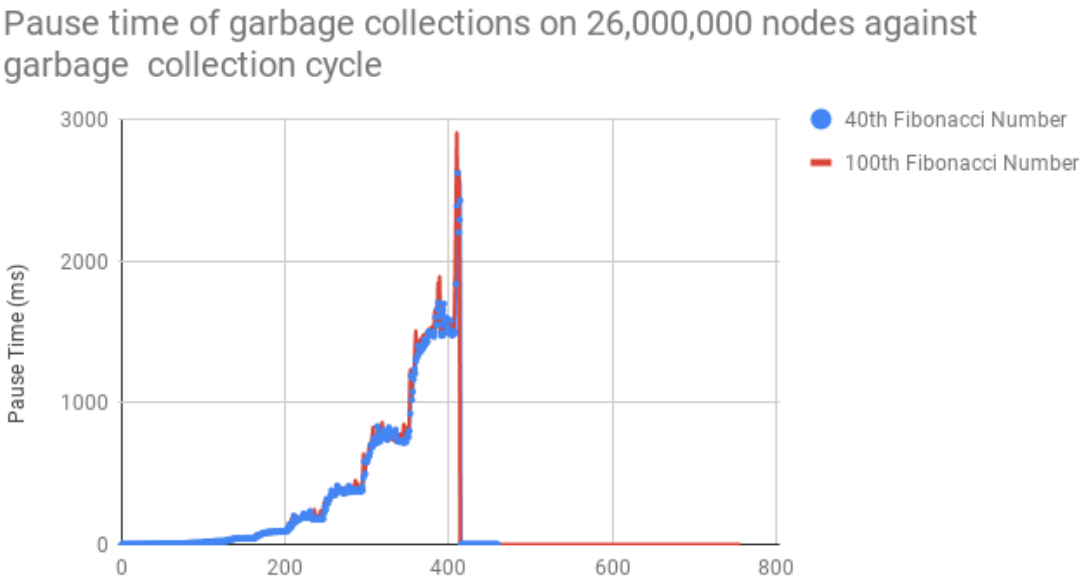
5.3.1 Trace Does Not Matter

When the object has large size, tracing time is a very insignificant portion of pause time comparing with the time of copy operation. We store a huge list with 26,000,000 nodes into a compact region. In the current compact region implementation in the

DDC, the reference list has 26,000,000 pointers to these nodes in the compact region since the "Cons" of the list is a boxed object. These objects will not be copied to the heap unless the compact region is marked dead. In another word, they will only be traced by the garbage collector at least once in each garabge collection cycle. We run the program ten times, and the aggregated data shows the following.



Initially, we think the peak is because of the garbage collection when computing the Fibonacci number and we were disappointed with the performance of using the reference list as the solution to the outgoing pointers. However, after changing the computation function, to calculate the 100th Fibonacci number, the tail of the graph suddenly gets longer and flat.



We realised the reference list did not cause the significant increase of pause time, however, the allocation of such large data structure into the compact region takes that time. When we implement the storeCR, we did not directly allocate the object into the compact region. Instead, we copy the object on the heap into the compact region and return the pointer to that object in the compact region. A further work may integrate the storeCR into the object creation so that an object can be directly allocated into the compact region.

The 26,000,000 nodes in the DDC are roughly 1GB. By comparing with the pause time in section 5.2.1, the first group result, 1G size, we realise to trace 1GB objects, which is a list containing 26,000,000 nodes, are faster than copying 1GB objects and the tracing time is even negligible. When the tracing time is negligible, the only factor that may cause the long-lived objects to slow down the garbage collection cycle is the copy operation. If an object in the compact region has reference to a heap object, it deduces the performance benefit of using the compact region for that particular object and the objects which garbage collector can reach from that object. Optimizations on this is described in the further work section.

Chapter 6

Related Work

6.1 Garabge-first garbage collection

Garbage-first garbage collector (G1) released 2004 [DFHP04] , and in 2018 it is the default garbage collection in Java 9. The G1 is the representative of the generational region-based memory management. In G1's model, the heap is split into numbers of small regions, and the region size is dependent on the heap size. For example, the 4GB heap may have 2MB region size. Like generational garbage collector, it has three types of regions by the lifetime of objects in regions. Objects are allocated to eden regions, and when a minor garbage collection happens, it copies the live object in the Eden region into the survivor region. If an object survives long enough, it is copied into the old region. Before the major GC happens, the G1 will concurrently mark live objects in old regions. When the mark finished, G1 starts a mixed GC which copies live old objects and compact them into other regions beside minor GC. G1 has enormous optimisations efforts, and it is widely used in production environments.

Compared with the compact region, G1 preserves complete automatic memory management and eliminate the pause time because of tracing live objects by introducing the concurrent marking. However, CPU still put the effort to trace the long-live objects on other cores. The G1 still needs to spend execution time to copy and compact

long-live objects in one old region to another. In the compact region system, we do not copy these object at all but we may have worse promptness if the programmer does not dissolve the compact region on time.

6.2 Direct Buffer

Off-heap memory is one of the general methods for storing large object and escape garbage collection cycle. It is used in various programming languages. For example, in Java, it is called the direct buffer. The direct buffer is claimed by calling the JVM unsafe function allocateMemory which calls malloc in most of the JVM implementation. The direct buffer address will preserve unchanged when the reference to the direct buffer is reachable so that it is primarily for the large and long-lived object when doing the native I/O operations. In the official JavaDoc, it recommends using the direct buffer when it "yields a measurable gain in program performance". When values having primitive types are written to the direct buffer, they are translated into a sequence of bytes. The interface to access the object in the direct buffer is limited to primitive types.

Like the compact region, direct buffer prevents garbage collector trace and copy long-live objects. The direct buffer has its benefit that it can be garbage automatically collected after it is not reachable. However, for value having abstract data types, it requires serialisation when write and deserialization when read. Compact regions, on the other hand, have a more clean interface and it does not need the programmer to implement serialisation and deserialization for the non-primitive data type. Overall, the direct buffer is not designed to store objects having references and interacting with objects in the heap.

6.3 Compact Normal Form

In chapter 2, we introduce the compact normal form (CNF) [YCA+15]. The compact normal form has similarity and difference with the compact region this thesis con-

tributes. To avoid ambiguity, we use CNF for the compact normal form and compact region or CR for compact region described in chapter 3.

The CNF uses a chain of memory blocks to represent a contiguous transmittable structure over distributed nodes. To share with other nodes, the data in the chain must have no outbound pointers to prevent dangling pointers. To avoid serialization cost, the transmitting data must be in a contiguous chunk of memory so that the networked representation is identical to the in-memory representation. In contrast, the compact region does not target the problem of sharing over distributed nodes so it can have mutability and allows the outgoing pointers into the compact region by modifying the garbage collector behaviours described in chapter 3.

The CNF and compact region both state the garbage collector will never trace into the "region". For the CNF, the reason behinds this is that evacuating the objects in the CNF will break the contiguous chain of objects, which introduce serialisation costs when transmitting the data to other nodes. Consider an example of Cheney's semi-heap garbage collector, if two root objects are the head of two different CNF chain, and they are evacuated to the to-heap during the garbage collection cycle, neither chain will be contiguous physically. However, for the compact region, the reason garbage collector does not trace in compact regions is to reduce the unnecessary trace and copying concerning long-lived objects.

Furthermore, the compact region needs to be dissolved by the programmer to reclaim the memory filled with long-lived or dead objects. The objects in the dead compact region are copied back to the heap by using the garbage collector then the compact region is deleted physically.

Chapter 7

Conclusion

7.1 Future Work

7.1.1 Production Tests

Recall that in section 5.1.2, we specify the evaluation of the compact region has only been done in the simulation tests. Potential further work will be to find some project which is already in production and having the problem with the long-lived object and implement compact region in the language as a library or build into the language and then attempts to optimise the project. One possible project that has the problem is the YourCraft Game Server project. Back in 2015, the YourCraft was famous Minecraft game server in China. The server has to keep too many states in the memory and many instances of the distributed nodes frozen for seconds because of the STW garbage collection. From the nature of the Minecraft server, we believe the compact region may apply to the server code design so that it could run more fluently.

7.1.2 Compact Region As A Build-In Feature

In section 5.3.1, we discuss the issue related to the storeCR implementation in the DDC. It copies the object from the heap into the compact region and then returns the pointer to the object in the compact region. This may cause issues when allocating a large number of objects in a non-tail recursive function. For example:

```
allocList {@r: Region}
    (cr : CR r)
    (curr : Nat )
    (max : Nat) : S (Write r) List
 = do case curr == max of
        True -> storeCR cr Nil
        False -> storeCR cr
                  (Cons curr
                   (countList cr (curr+1) max))
```

All the thunk and the allocated list will remain in the heap until the first function call finishes. One possible solution to this problem is to store the object into the compact region directly. It may require the compact region to be a built-in functionality of the language so that when allocating objects, the compiler distinguish two types of allocation, allocating on the heap and allocating on the compact region.

7.1.3 Reducing Copy Operations

In section 5.3, we discuss the outgoing pointers. A programmer may intentionally set a field of an object in the compact region to refer to an object on the heap. The object on the heap can be huge so that it may cause significant pause time since the copying of that large object on the heap. It may be an unexpected behaviour to the programmer who trusts and uses the compact region. One solution to this problem is that the garbage collector also automatically copy that object on the heap into the compact

region so that the compact region can be self-contained after a garbage collection cycle. It may create problems, such that waste of the memory if the programmer changes the references too frequently. One solution to that is to implement the proxy object described in chapter 3. When the Ref object in the compact region is updated to point to other objects, the old object it points to will be copied back to the heap, and leave a forward object in the compact region. Reclaim the forward object after compact region garbage collecting phase.

7.2 Summary of Contributions

We mainly solve the problem of the garbage collection pause time caused by long-lived objects in the heap by contributing:

- The description of the compact region user interface and allows mutability and outgoing pointers by using the reference list (chapter 3.1 section 1)
- The detailed explanation of garbage collector modifications to integrate the compact region into a language runtime system. (chapter 3.1 section 2)
- An implementation of the compact region in DDC runtime system and corresponding garbage collector modifications to a copy garbage collector based on Cheney's algorithm.
- Test results of programs simulating real-life program which may have issues with long-lived objects and discovered the cost of tracing objects may be insignificant to the time of copying objects.

Bibliography

[BCR04] David F. Bacon, Perry Cheng, and V. T. Rajan. A unified theory of garbage collection. *ACM SIGPLAN Notices*, 39(10):50, Jan 2004.

[BR01] David F. Bacon and V. T. Rajan. Concurrent cycle collection in reference counted systems. *ECOOP 2001 Object-Oriented Programming Lecture Notes in Computer Science*, page 207235, 2001.

[BTV96] Lars Birkedal, Mads Tofte, and Magnus Vejlstrup. From region inference to von neumann machines via region representation inference. *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL 96*, 1996.

[Cha87] David R. Chase. Brief survey of garbage collection algorithms. 1987.

[Che70] C. J. Chenney. A nonrecursive list compacting algorithm. *Communications of the ACM*, 13(11):677678, Jan 1970.

[Col60] George E. Collins. A method for overlapping and erasure of lists. *Communications of the ACM*, 3(12):655657, Jan 1960.

[DFHP04] David Detlefs, Christine Flood, Steve Heller, and Tony Printezis. Garbage-first garbage collection. *Proceedings of the 4th international symposium on Memory management - ISMM 04*, 2004.

[DH99] Sylvia Dieckmann and Urs Hlzle. A study of the allocation behavior of the specjvm98 java benchmarks. *ECOOP 99 Object-Oriented Programming Lecture Notes in Computer Science*, page 92115, 1999.

[gdd] Discus 0.5.1 documentation.

[Hay91] Barry Hayes. Using key object opportunism to collect old objects. *Conference proceedings on Object-oriented programming systems, languages, and applications - OOPSLA 91*, 1991.

[HET02] Niels Hallenberg, Martin Elsman, and Mads Tofte. Combining region inference and garbage collection. *ACM SIGPLAN Notices*, 37(5):141, 2002.

[HW67] B. K. Haddon and W. M. Waite. A compaction procedure for variable-length storage elements. *The Computer Journal*, 10(2):162165, Jan 1967.

[JHM12] Richard Jones, Antony Hosking, and Eliot Moss. *The garbage collection handbook: the art of automatic memory management*. CRC Press, 2012.

[Lan17] The Discus Language. Source file of ddc runtime objects code version 0.5.1, 2017.

[LG88] J. M. Lucassen and D. K. Gifford. Polymorphic effect systems. *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL 88*, 1988.

[Lip10] Ben Lippmeier. Type inference and optimisation for an impure world. May 2010.

[Ma12] Tran Ma. type-based aliasing control for the disciplined disciple compiler, 2012.

[Mcc60] John Mccarthy. Recursive functions symbolic expressions and their computation by machine, part i. *Communications of the ACM*, 3(4):184195, Jan 1960.

[SM06] Memory management in the java hotspot virtual machine, 2006.

[TGS08] Franklyn A. Turbak, David Gifford, and Mark A. Sheldon. *Design concepts in programming languages*. MIT Press, 2008.

[TJ92] Jean-Pierre Talpin and Pierre Jouvelot. Polymorphic type, region and effect inference. *Journal of Functional Programming*, 2(03):245271, 1992.

[TT97] Mads Tofte and Jean-Pierre Talpin. Region-based memory management. *Information and Computation*, 132(2):109176, 1997.

[Ung84] David Ungar. Generation scavenging. *Proceedings of the first ACM SIGSOFT/SIGPLAN software engineering symposium on Practical software development environments - SDE 1*, 1984.

[YCA+15] Edward Z. Yang, Giovanni Campagna, mer S. Aacan, Ahmed El-Hassany, Abhishek Kulkarni, and Ryan R. Newton. Efficient communication and collection with compact normal forms. *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming - ICFP 2015*, 2015.

Appendix 1

This appendix introduces the steps to get the compact region implementation in DDC up and running.

A.1 Getting Started

The first step is to set up the DDC compiler. The instruction is on the webpage http://discus-lang.org/section/01-GettingStarted.html [gdd].

A.1.1 Prerequisites

The DDC requires three dependencies to install. We suggest using recent versions of all dependencies.

- GHC
- Cabal
- LLVM

When this thesis is written, we use GHC version 8.2.1 and Cabal version 2.2.0.0 and LLVM version 5.0.0.

A.1.2 Clone Code form GitHub

The code is held on the GitHub and we have two versions of the code.

- https://github.com/JHXSMatthew/ddc/tree/compactRegionDev
- https://github.com/discus-lang/ddc

The former version is the implementation when this thesis is written and the other version is the latest DDC code.

To clone it, use

```
git clone https://github.com/discus-lang/ddc
cd ddc
```

A.1.3 Compile DDC

To install dependencies

```
make setup
```

To build the compiler

```
make
```

A.1.4 Compact Region

The Compact region implementation is mostly in the runtime system and it includes

- module: src/s2/ddc-runtime/salt/runtime64/primitive/CRegion.dcs
- Garbage Collector: src/s2/ddc-runtime/salt/runtime/Collect.dcs

And an interface to source language

- ddc/src/s2/base/Data/CompactRegion.ds

Tests to demostrate the features of compact regions are:

- Basic Demo: test/ddc-demo/source/Discus/30-Library/01-Data/07-CR/01-Base
- CRT Demo: test/ddc-demo/source/Discus/30-Library/01-Data/07-CR/02-HitLimit
- GC Demo: test/ddc-demo/source/Discus/30-Library/01-Data/07-CR/03-GC

To compile and run the test, use

```
./bin/ddc test/ddc-demo/source/Discus/30-Library/01-Data/07-CR/01-Base/Mai
```

Tests to demostrate the performance of compact region on long-lived objects are in

- test/ddc-demo/source/Discus/30-Library/01-Data/07-CR/99-skip

These are tests we described in the evaluaion section.