Conservative Garbage Collection in Kernel and Mutator Space

BY GREGOR BACHINGER

Bachelor's thesis submitted in partial fulfilment of the requirements for the degree of Bachelor of Science in Computer Science.

Supervisor: Univ.-Prof. Dr.Ing. Christoph Kirsch

Salzburg, September 2020

Abstract

Compiling C or any similar language into binaries only provides support for explicit, manual memory management. To combat these issues, garbage collectors have been employed which handle memory management automatically. Since garbage collectors are specialized and use a lot of assumptions about their target environments, different algorithms are usually used for collectors in and outside of the mutator's address space. We have designed and implemented a conservative garbage collector that can run either in kernel or mutator space, or even in both at the same time. Running our collector in kernel and mutator space at the same time shows, that the collector is able to collect its own garbage. Furthermore, our experiments show the runtime/space tradeoff between collector invoke frequency and running it on machine and application level.

Keywords: Garbage Collection, Reachability Analysis, Conservative Garbage Collection, Mark/Sweep Collector, Selfie, Collection in Mutator and Kernel Space

Table of Contents

1 Introduction 4

1.1 Problem Definition 4

1.2 Contribution 4

1.3 Garbage Collection Primer 4

1.3.1 Principles 4
1.3.2 Strategies 5
1.3.3 Conservative vs. Precise Garbage Collection 5

1.4 The Target Architecture 5

1.4.1 CPU 6
1.4.2 Memory 6
1.4.3 Function Calling Convention (Stack) 7
1.4.4 Availability 8

2 The Conservative Garbage Collector Implementation 8

2.1 The Interface 8

2.2 Metadata 9

2.3 Initialization of the Garbage Collector(gc_init) 10

2.4 Garbage Collection 10

2.4.1 Mark Requirements 10
2.4.2 Mark 11
2.4.3 Sweep 12
2.4.4 Visualization of Mark and Sweep 13
2.4.5 When to Collect 15

2.5 Memory Allocation 15

2.5.1 Creating New Memory 15
2.5.2 Reusing Free Memory 16

3 Two Variants of Deployment 17

3.1 Library 17

3.1.1 Implementation 17
3.1.2 Advantages 18
3.1.3 Disadvantages 19

3.2 Syscall 19

3.2.1 Implementation 20
3.2.2 Advantages 21
3.2.3 Disadvantages 21

3.3 Code Sharing between Library and Syscall 21

4 Garbage Collection in Selfie 22

4.1 Library vs. Syscall 22

4.2 Self-Collecting Garbage Collection 23

4.3 Garbage Collection at Boot Level 0 23

5 Analysis of the Algorithm 24

5.1 Collection 24

5.1.1 Mark 24

5.1.2 Sweep 24
5.1.3 Summary 24
5.2 Reusing Memory 25
5.3 Allocating New Memory 25
5.4 Relation between Time and Space Complexity 25
6 Experiments 26
7 Optimization 30
7.1 When to Collect 30
7.2 Improving Reachability Analysis 30
7.3 Reusing, Fitting and Free List Improvements 30
8 Conclusion 31
9 Bibliography 32

1 Introduction

1.1 Problem Definition

By itself, C or similar languages do not provide any automatic memory management. The mutator has to avoid issues like dangling pointer and memory leaks. Therefore, garbage collectors are used to handle memory management automatically. Garbage collectors are highly specialized and use a lot of knowledge and assumptions about their target environment. Furthermore, different algorithms are usually employed to perform collection in and outside of the memory.

1.2 Contribution

We have designed and implemented a conservative garbage collector that can run either in kernel or mutator space, or even in both at the same time. Our collector does not only work in an uncooperative environment (i.e. inside the mutator's address space) but also in the implementation (i.e. as kernel extension) of a cooperative environment. Running both variants of the same collector enables it to even collect itself. Furthermore, this thesis analyzes runtime and space complexity tradeoff between the invoke frequency of the collector and also features a comparison of both deployment versions.

1.3 Garbage Collection Primer

A garbage collector attempts to reclaim unreachable/dead memory (i.e. memory which can no longer be accessed by the program). The benefit of using a garbage collector is the simplification of memory management, as manual memory management might lead to bugs caused by the software's developer. Garbage collectors typically consist of two parts: An allocator and a collector.

Garbage collectors perform automatic dynamic memory management through the following operations [1]:

- Allocate/give back memory from/to the operating system
  - i.e. the garbage collector's memory pool
- Hand out memory to the application
- Perform reachability analysis (i.e. is live memory still reachable?)
- Reclaim unreachable memory

1.3.1 Principles

Basically, all memory that cannot be accessed in the future must be reclaimed. To do that, garbage collectors can be deployed in different ways. They can either be a part of the language specification (e.g. Java [2]) or an extension to manual memory management (i.e. co-existence of garbage collected and manually managed memory). Furthermore, the collector must be closely integrated with its and the system's memory allocator. This thesis shows an algorithm which can be used both as part of the system (like the JVM collector) and as an extension (like [3]).

1.3.2 Strategies

There are many different strategies to perform garbage collection but almost all algorithms are one of tracing or reference counting. Tracing uses reachability analysis to determine whether an object can be reclaimed or not. Different methods can be used to achieve this (e.g. mark/sweep). The alternative method - reference counting [4] - stores information about every pointer (i.e. a counter). If a certain condition (i.e. the counter reaching 0) is satisfied, the pointer is freed. There are also escape analysis (e.g. [5]) and timestamp & heartbeat methods. The former of which is usually not a full garbage collector and the latter aren't traditional garbage collection methods.

1.3.3 Conservative vs. Precise Garbage Collection

Further generalizing, collectors can be either conservative or precise. The former working without compiler cooperation, which means that they have to discover all pointers by themselves. This leads to problems, as pointers and integers look the same from a compiler's point of view. Therefore, every integer must be treated as a potential pointer. Furthermore, all mutators need to stop until garbage collection has been completed. Precise garbage collectors on the other hand use gc-maps which assume knowledge of pointers. The algorithm specified in this thesis is conservative.

1.4 The Target Architecture

Even though conservative garbage collection works in an uncooperative environment [3], knowledge about the target system's architecture is needed. This project uses RISC-U as its instruction set, for reasons of simplicity. "RISC-U is a tiny subset of the 64-bit RISC-V instruction set." [6] RISC-V provides a CPU with 32 general-purpose 64-bit registers as well as a program counter. The whole machine is represented using the von Neumann model (i.e. CPU connected to memory via memory bus, see Figure 1).

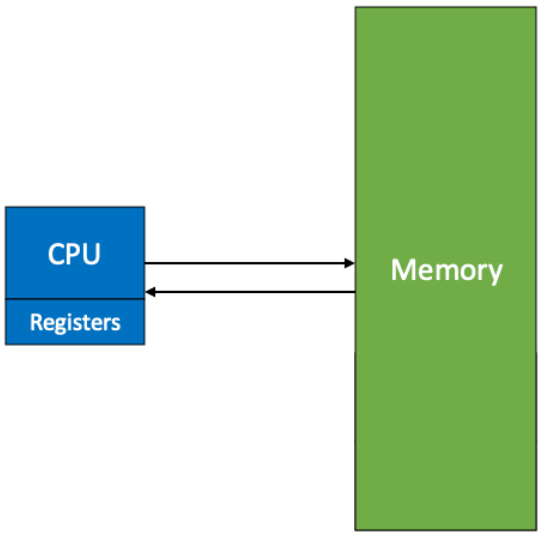

Figure 1: von Neumann model (simplified)

1.4.1 CPU

RISC-U consists of six types of instructions: initialization (load upper/add immediate), memory (load and store), arithmetic (add, subtract, multiply and divide/remainder unsinged), compare (set less than unsigned), control (branch on equal, jump and link (register)), system (environment call - ecall). Since RISC-U is limited to 64-bit registers, datatypes are also limited to 64-bit integer (uint64_t) and pointer to integer (uint64_t*). Therefore, a valid address must always be a multiple of 8 bytes.

1.4.1.1 Registers

As already mentioned, the RISC-U architecture has 32 general-purpose 64-bit registers. These are as follows [7]:

- ZR – zero
- RA – return address
- SP – stack pointer
- GP – global pointer
- TP – thread pointer
- T0-T6 – temporary
- S0-S11 – saved register
- A0-A7 – function argument

The zero register is immutable and always zero. Thread pointer, S1-S11 and A4-A5 are unused. A1-A3 and A6-A7 are usually not used, with the only exception being some system call implementations as specified by the RISC-V pk [8]. A0 is used as return value, RA as return address, SP as stack pointer, GP as global pointer and S0 as stack frame pointer. T0-T6 are used as temporary registers.

1.4.2 Memory

Consider a 2^{32} byte word aligned (64-bit words, see 1.4.1 CPU), byte addressed linear address space. This address space is divided into four segments: code, data, heap and call stack. Code and data segment are static and determined at compile time, while stack and heap are dynamic (i.e. runtime dependent). The order of these segments is shown in Figure 2.

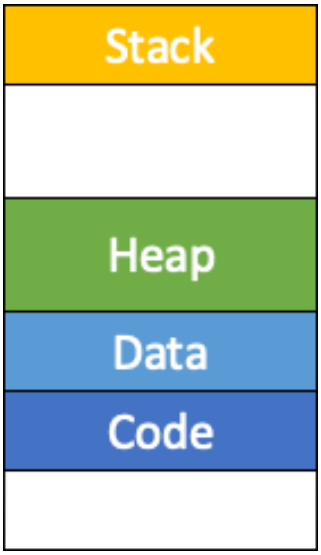

Figure 2: memory - segment order

Code and data segment are placed on the bottom side of the memory with an offset of 2^{16}, meaning the code segment always starts at address 65536 and the data segment starts at $65536 + code\_segment\_size$. This conforms to the entry point 0x1000 according to the RISCV pk [6]. The code segment contains instructions and the data segment contains all global variables. These two sections round up the static portion of the memory. Above it lies the dynamic section. This border point between static and dynamic memory is called program break and calculated using Equation 1.

$$program\_break = 65536 + code\_segment\_size + data\_segment\_size$$

Equation 1: program break

Heap and stack make up the dynamic part of the memory. The heap resides just above the program break and grows upwards. It holds all dynamically allocated memory (i.e. memory allocated by malloc(…)). In general, every kind of allocator could be used to create heap memory. This memory model assumes the usage of a simple bump pointer allocator. The stack is the second segment which resides on the upper bound of the memory and grows downwards.

1.4.3 Function Calling Convention (Stack)

According to [7], the RA, T0-T6 and A0-A7 registers are not preserved across calls. In order to call functions certain registers must be saved by the caller. This is handled using a procedure/function prologue.

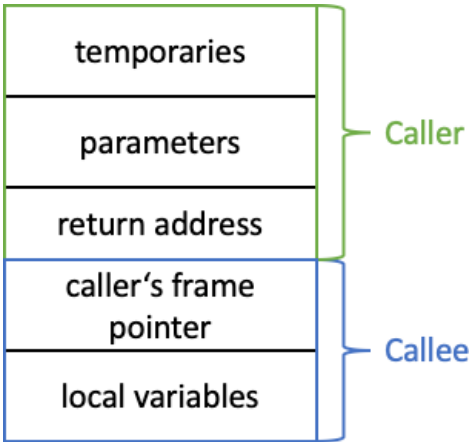

Figure 3: call stack frame

First of all, the temporaries are saved (if temporaries are in use at call) by pushing them onto the call stack. Next up are parameters and afterwards the caller's return address (stored in RA) and the callee's frame pointer (stored in S0). Last but not least, memory for the callee's local variables is allocated. Figure 3 shows the structure of this call frame. Since A1-A7 are not used for a standard function call, they can be ignored.

1.4.4 Availability

This architecture is implemented by the Selfie project. The Selfie Project provides an educational platform for teaching the design and implementation of programming languages and runtime systems. It offers a compiler, emulator and hypervisor which are all used for this project. The programming language used by Selfie is called C* which is tiny a subset of C. [6] There are even more features, however those are irrelevant to this thesis.

2 The Conservative Garbage Collector Implementation

This section contains information about the actual implementation of the garbage collector. The goal of this project is to keep the collector as simple as possible. Keep in mind, that this work is more of a proof of concept and for educational purposes as opposed to a system used in production.

2.1 The Interface

To use this garbage collector, the mutator needn't consider anything apart from using the right functions. There are different variants of this garbage collector, which are talked about in 3 Two Variants of Deployment. For now, it is assumed, that the garbage collected allocator is invoked using the gc_malloc function (see Code block 1).

```
gc_malloc(size):
        IF should_collect:
                collect()
        memory = get_free_memory()
        IF memory != NULL:
                return memory
        ELSE:
                return allocate_new_memory()
```

Code block 1: gc_malloc

Before taking a closer look at this function itself, it needs to be noted that new memory is always a combination of metadata and objects.

2.2 Metadata

When allocating heap memory, information about the allocated object (i.e. address, size, etc.) needs to be stored, in order to collect them later. This information is referred to as metadata and can be observed in Figure 4.

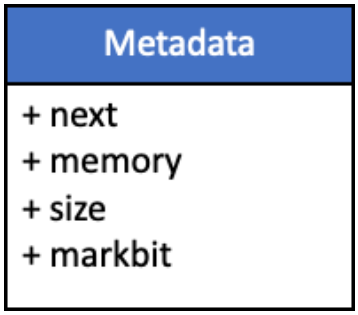

Figure 4: metadata entry

First of all, the entries are managed as singly linked lists (see Figure 5). This is not optimal in terms of runtime complexity (see 5 Analysis of the Algorithm). The first pointer (field 1) points to the next metadata entry of the list it belongs to. If there is none, it is simply set to zero. Following that, address and the size of the object itself are stored in field 2 and 3 respectively. Last but not least, a mark bit is stored in field 4 which is used to perform mark/sweep collection (see 2.4.2 Mark, 2.4.3 Sweep).

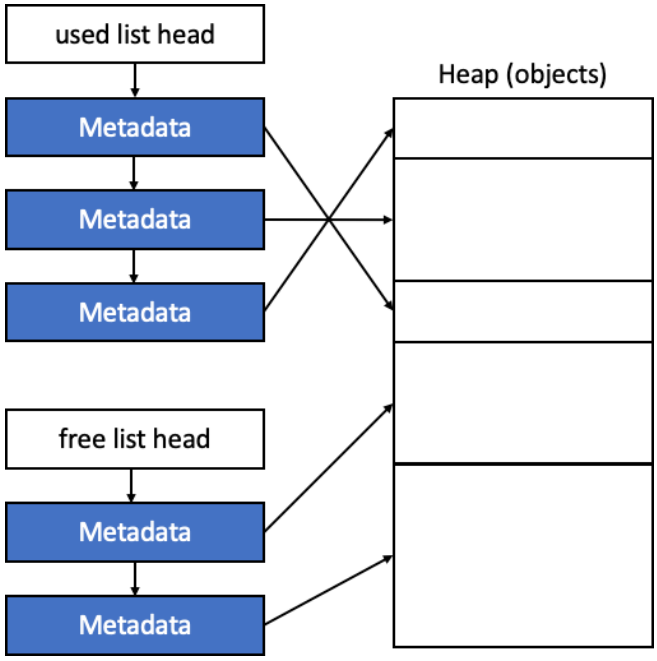

Figure 5: metadata - object relation and lists

2.3 Initialization of the Garbage Collector(gc_init)

Before any garbage collected memory can be allocated, some prerequisites must be fulfilled. These include: Setting up metadata lists and determining heap/data segment bounds (i.e. start and end of segments). Heap start and end are simply set to the current program break, while the data segment must be calculated using data segment size and global pointer.

There are two metadata lists: a “used list” and a “free list” (see Figure 5). The used list stores all metadata entries which reference live (i.e. reachable) memory. The free list on the other hand stores all metadata entries which reference dead (i.e. unreachable) memory. These lists are initialized by setting their pointers to zero (i.e. list empty).

Last but not least, a flag is set to indicate that the collector has been initialized.

2.4 Garbage Collection

The garbage collector is conservative and implemented in terms of a classic mark/sweep collector. This section talks about the detailed implementation of mark and sweep as well as their requirements.

2.4.1 Mark Requirements

During the mark phase, all relevant sections in memory are traversed to find out if a heap address can be reached. Before taking a closer look at the actual mark implementation, those sections need to be defined. In order for memory to be accessible, it needs to be pointed to by a variable. Variables can be one of three types: intermediate value (i.e. stored in a relevant register during calculation), local (residing in stack) or global (residing in data segment). These locations are referred to as root segments. Reachability analysis needs to be performed for each word stored in those.

Selfie's function calling convention (see 1.4.3 Function Calling Convention (Stack)) eliminates the need to scan registers. This is because: Only ZR, RA, SP, GP, T0-T6, A0 and S0 are usually (A1-A3, A6-A7 in system calls) used. ZR, RA, SP, GP and S0 cannot contain a valid reference to an object. Furthermore, when collect is invoked A0 is assumed to never contain any valid reference to an object and the temporary stack is empty (since temporaries are pushed onto the call stack beforehand). Therefore, the registers will never contain any valid reference to an object and can safely be ignored during marking. This leads to the only root segments being call stack and data segment. In order to traverse those, their bounds have to be determined.

Data segment bounds can be determined quite easily: During compilation the size of the data segment is saved. In order to fetch this value during runtime, a library function is emitted which returns the data segment size. However, since library functions in Selfie are emitted before code compilation but the data segment size is only known after code compilation, a fixup needs to be performed:

- Emit library function interface (i.e. emit library function as usual without data segment value (replace with NOP instruction), save address of NOP instruction)
- Compile code
- Perform fixup (i.e. use saved address to replace NOP instruction with actual data segment size)

The second data segment bound fetcher yields the global pointer of the application (i.e. return GP). In combination, those functions can be used to determine data segment start and end:

$$ds\_end = global\_pointer$$

Equation 2: data segment end

$$ds\_start = ds\_end - ds\_size$$

Equation 3: data segment start

Stack bounds are easier to determine than data segment bounds. The upper bound is always 2^{32} and the lower bound is stored in the SP register. To fetch this lower bound, a library function similar to fetch the global pointer is introduced.

2.4.2 Mark

As already mentioned, conservative garbage collection considers every integer as a potential address. The mark process starts by scanning the root segments and performs reachability analysis for each of their words. For each valid object the mark process is performed recursively.

```
mark_segment(lower_bound, upper_bound):
        address = lower_bound
        WHILE address < upper_bound:
                word = get_word_at(address)
                IF is_valid_heap_pointer(word):
                        mark_object(word)
                address = address + size_of_word
```

Code block 2: mark segment

```
mark_object(word):
        metadata = get_metadata_of_address(word)
        IF metadata.markbit == UNREACHABLE:
                metadata.markbit == REACHABLE
                address = get_address_of(metadata)
                WHILE address < get_address_of(metadata) + get_size_of(metadata):
                        mark_object(address)
                        address = address + size_of_word
```

Code block 3: mark object

Reachability analysis is performed as follows: First of all, a basic sanity check is performed (i.e. is it a multiple of 8 (see 1.4.1 CPU) and is it a value in $[0; 2^{32}]$ (see 1.4.2 Memory)). After that a heap bounds check is performed, . This is an attempt to weed out all invalid addresses before the actual used list search is carried out (since conducting a full list search of an invalid address is way more expensive than these small sanity checks, see: 2.4.2 Mark). If all checks passed, the list search is executed. Upon success (i.e. list search returns non null pointer) the object is marked (i.e. the returned metadata's markbit is set) and mark_object is called recursively. It has to be noted though, that an object is never marked twice (i.e. markbit set → skip recursive marking).

One last thing about the mark phase: Usually, there needs to be a prepare mark phase (which sets all markbits to zero) before marking. This phase can be moved to the sweep phase to avoid having an extra list iteration.

2.4.3 Sweep

The mark phase alone merely determines the reachability of all currently in use objects. But to actually be able to collect unused memory a sweep phase, based on the mark phase's results needs to be carried out:

```
Sweep():

node = get_used_list_head()
WHILE node != NULL:
        IF node.markbit == UNREACHABLE:
                free(node)
        ELSE:
                node.markbit = UNREACHABLE
        node = node.next
```

Code block 4: sweep

Each metadata entry whose object is unreachable is freed. Freeing an object is basically moving its metadata from the used list to the free list. This is optional though and the collector provides the option to not reuse memory, meaning that instead of moving the metadata from list to list, it's simply removed from the used list. The consequence caused by moving an object out of the used list is that a pointer to it would no longer pass the last reachability check.

2.4.4 Visualization of Mark and Sweep

Consider the structure shown in Figure 6. On the left side, the root segment is visualized consisting of the data and stack segment's words. It can be observed, that there are only two valid pointers.

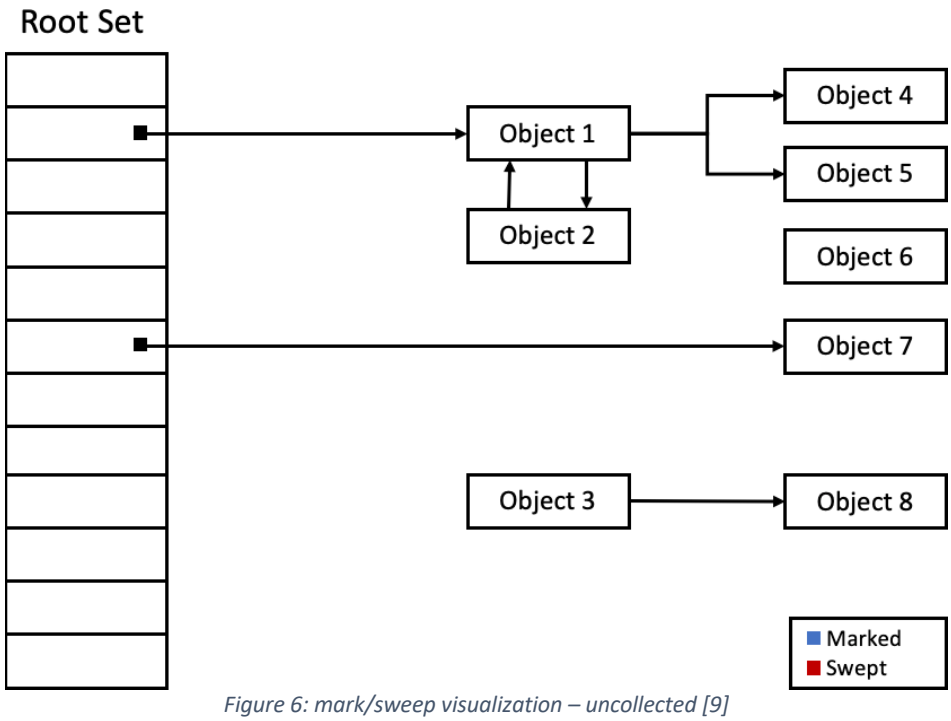

Figure 6: mark/sweep visualization – uncollected [9]

As a result, mark segment only affects these two objects (see Figure 7). Both of these objects will be marked recursively, though only Object 1 contains valid references.

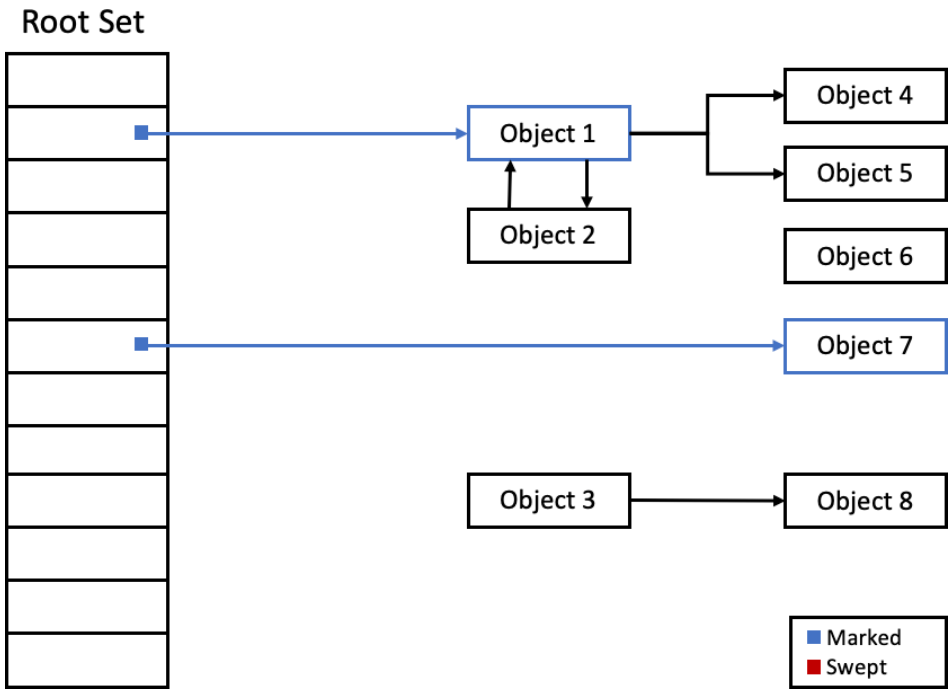

Figure 7: mark/sweep visualization - mark segment [9]

Looking at the results of a recursive mark object (of Object 1), it can be observed (Figure 8) that all reachable objects have been marked successfully. While Object 2 would actually reference a valid object, the recursive mark will not be carried out since this object has already been marked, therefore avoiding infinite loops.

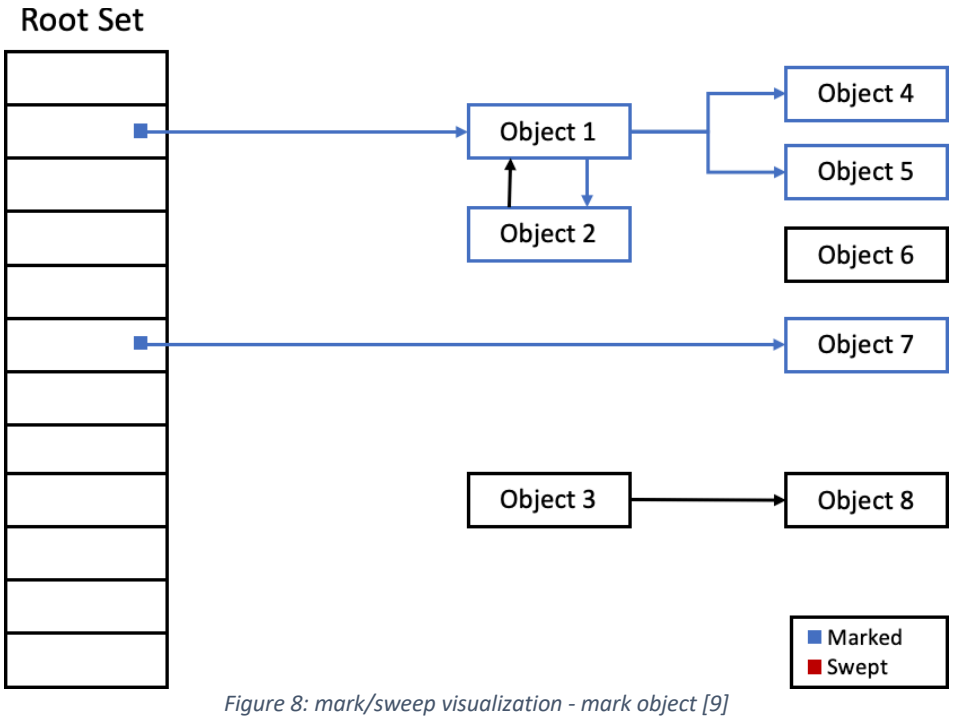

Figure 8: mark/sweep visualization - mark object [9]

After marking, the sweep phase is invoked, and all unmarked objects (Figure 9) are freed (or dropped if there is no memory reuse). This rounds up the whole collection process.

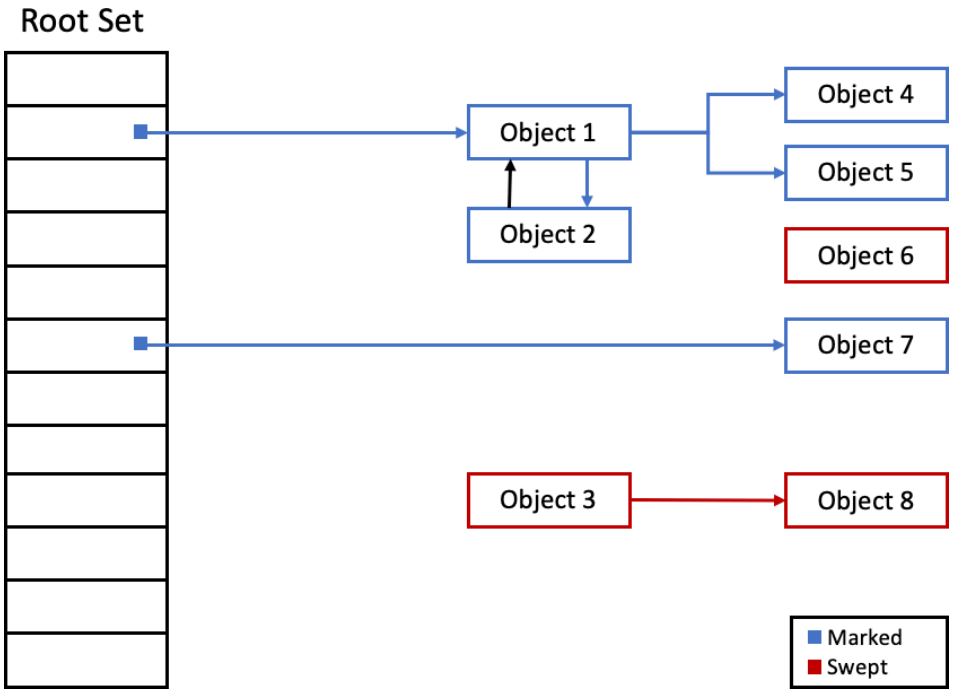

Figure 9: mark/sweep visualization - swept (collect complete) [9]

The impact of conservatism as stated by [3] is, that memory is not lost as a result of the collection strategy. On the other hand, false positives are not only possible but with growing application complexity very likely. Furthermore, performing collection like that is very expensive and should not be done before every allocation.

2.4.5 When to Collect

To ensure that the surplus of allocated memory is minimal, the collector would need to collect before every allocation. Since collection is so expensive however, there needs to be some metric telling the collector when to collect. There are many different ways to implement a metric like that ranging from simple and dumb to complex and smart. This project's collector uses a very simple metric: A collect period constant. This means there is a counter counting the number of allocations. If this counter reaches the collect period, it will be reset, and garbage collection triggered. This is neither optimal in terms of space nor time complexity, however it provides a deterministic way to investigate time/space tradeoffs. For an in-depth analysis of this metric and possible improvements, see 7.1 When to Collect.

2.5 Memory Allocation

A garbage collector consists of two components: a collector and an allocator. Up until now, only the collector has been specified. This section covers how and when memory is created and reused.

2.5.1 Creating New Memory

To allocate a new object, memory is requested from the system's allocator and moved to the pool of the garbage collector. The pool consists of all objects whose metadata exists and is in one of used or free list (see Figure 10). Selfie specifies this base allocator to be a bump pointer allocator. This is a simplification, since only a single continuous heap is assumed (see 1.4.2 Memory). In theory, any type of allocator could be used as base allocator. Currently, the

reachability analysis actually supports a fragmentary heap (because of the list search), however this might be subject to change.

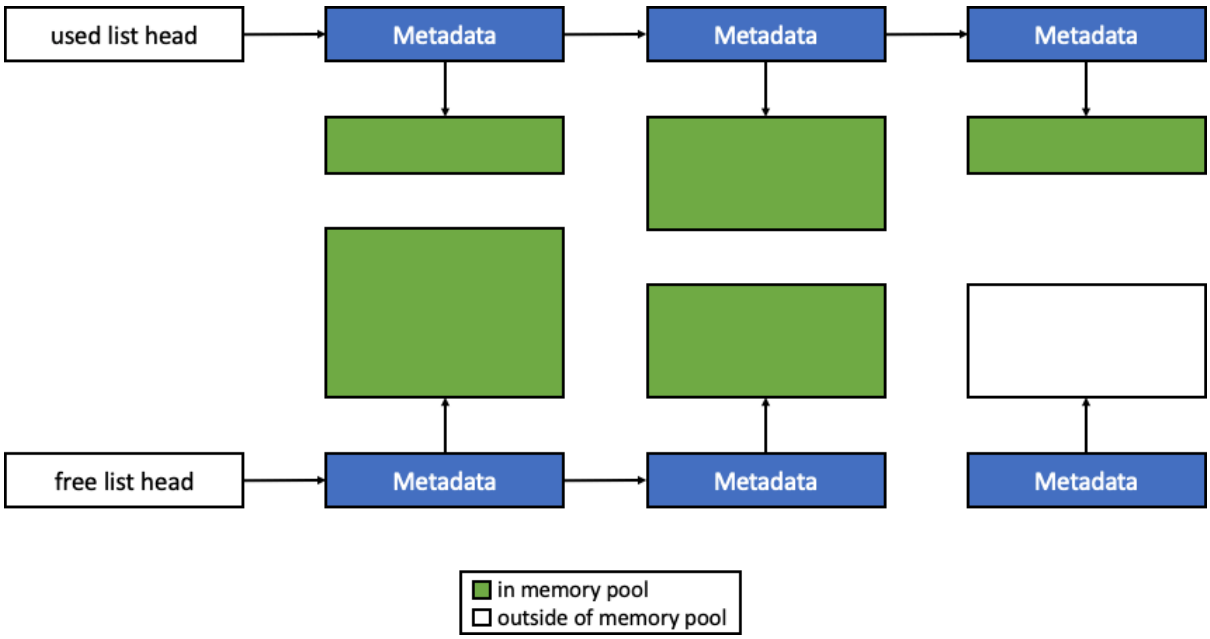

Figure 10: visualization of garbage collector's memory pool

As already mentioned, not only object but also metadata memory must be allocated. Metadata can be stored in different locations and even different address spaces (see 3.2 Syscall). Base allocation calls must be checked for validity and error handling of the garbage collector's allocator must conform with the base allocator's error handling (e.g. return null pointer, throw exception, etc.). Other than that, nothing special needs to be kept in mind – simply allocate metadata and object, set the metadata's address and size fields, push newly allocated entry to the new list and return the pointer to the object.

2.5.2 Reusing Free Memory

The ability to reuse memory is optional and may not be performed. If reusing is turned off, the free list will not be searched for reusable entries and objects are always created anew. Otherwise the free list is traversed to try and find a suitable object. The approach used is a modified first fit algorithm (i.e. find first entry with exact size).

While this is obviously harmful to both space and time complexity, it is the simplest way to minimize the number of metadata entries. Free memory is kept track off using metadata as free list entries. This kind of application obviously wouldn't need a markbit but creating a different type of entries for the free list (and translating a used list entry to a free list entry) would just make the algorithm more complex. Furthermore, they would need to be translated back when reusing their objects.

If an entry is found, it is simply moved back to the used list. Afterwards, the object's pointer is returned. If no suitable entry in the free list is found, a new one is created (see 2.5.1 Creating New Memory). One last thing about memory reusing: Memory may or may not be zeroed

when reusing it. While not zeroing the reused memory is obviously better in terms of performance, it might lead to false positives during mark's reachability analysis.

For example: Consider two free objects. During its lifespan object 1 referenced object 2. If reused memory is not zeroed and object 1 would be reused, the memory of object 1 would still reference object 2. Upon performing reachability analysis, a wrong positive would occur, since the reused object 1 still wrongfully references object 2.

3 Two Variants of Deployment

This section's aim is to describe the use cases, implementation and differences of two garbage collector deployment variants. The former is the classic approach of a garbage collector library, while the latter describes garbage collection as a kernel/hardware feature, which is referred to as syscall version.

3.1 Library

The library version of the garbage collector performs garbage collection in the same address space as the mutator (i.e. the garbage collected allocator and the system's actual allocator are co-existing). Furthermore, the mutator can decide whether they want to allocate collected or uncollected memory. Both types of memory can exist in the same heap without any problems. Figure 11 provides a visualization of this concept.

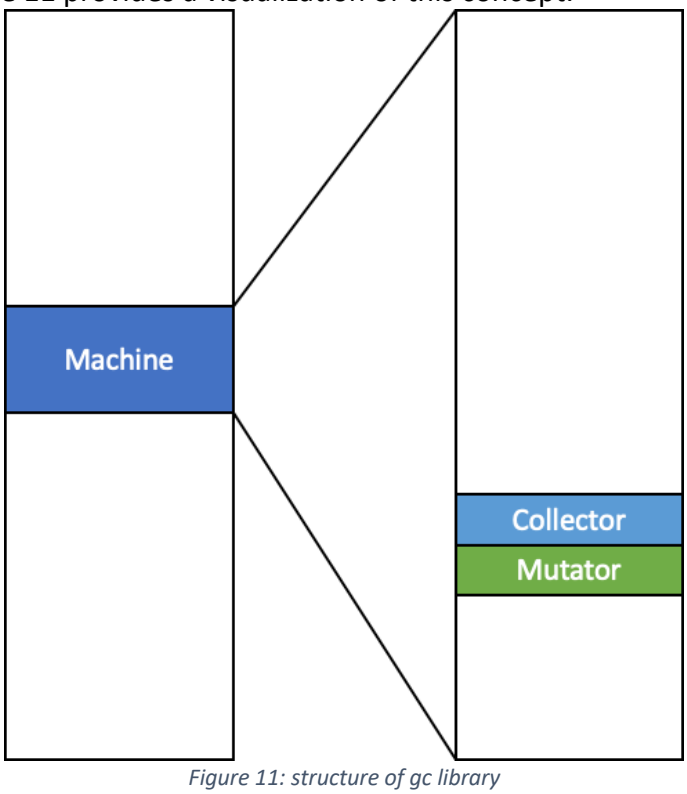

Figure 11: structure of gc library

3.1.1 Implementation

Syscall and library variant differ in four ways: bounds fetcher, metadata management, memory allocation and calling convention.

First of all, let's consider the bound fetcher functions of the library. As already mentioned, a total of six different bounds (data segment's, call stack's and heap's start/end) must be determined. Data segment end and stack start are stored in the GP and SP register respectively. Since C* code does not have explicit access to the registers themselves, library functions are used as stated in 2.4.1 Mark Requirements. The data segment start is calculated using the provided data segment size and program break fetcher. The stack end is a constant (i.e. $VIRTUALMEMORYSIZE = 2^{32}$). Heap bounds need to be tracked using the gc_malloc function itself. In essence the start will be determined upon initializing the garbage collector, while the end is updated after every new collected allocation.

Aforementioned library functions require a specialized compiler. In addition to all requirements mentioned in 1.4 The Target Architecture, it also needs to support a flag which generates these functions. The "-gc" flag has been introduced as a universal way to turn the garbage collector on and off. In this case it indicates whether or not to generate these bound fetcher library functions.

Up until now, only the basic prerequisites have been fulfilled. Compiling any application with the flag alone would not include the actual garbage collector in the binary. Luckily Selfie provides an in-memory linker to easily include the garbage collector. Since the garbage collector library itself is contained in Selfie's source code linking is as simple as compiling the application together with selfie.c. Keep in mind, that selfie.c contains a main function which needs to be renamed in order to avoid redefinition (see: [6] → Makefile targets: selfie.h and gclib). Note: The bound fetcher functions are also contained in selfie.c, however they always return 0 if the gc flag has been set. This enables checking whether or not the library is available (i.e. application has been compiled using the "-gc" flag).

Next up, metadata management needs to be considered. The library performs garbage collection on application level, meaning that metadata needs to be stored inside of the application's memory. One option would be to split the heap into two parts, one containing only object and the other containing only metadata memory. If this object section was guaranteed to only contain objects, reachability analysis could be reduced to a simple bound check (as opposed to a list search). To keep the collector simple however the heap is not split, meaning that metadata, objects and even uncollected memory use the same, shared heap.

Since all types of allocated memory share the same heap, the base allocation can be limited to a single allocator (i.e. Selfie's bump pointer allocator). This leads to both metadata and objects using the malloc function to allocate memory.

To allocate collected memory, the mutator needs to substitute malloc with gc_malloc.

3.1.2 Advantages

Once compiled, a binary using the garbage collector's library variant can be executed on any RISC-U (and therefore RISC-V) machine. It is also very independent of Selfie (apart from the mandatory library functions) and could be ported quite easily. As opposed to the syscall variant, the library can also use both collected an uncollected memory. However, this could be seen as both an advantage (i.e. enable more freedom) and a disadvantage (i.e. bugs are more likely to occur).

3.1.3 Disadvantages

The most apparent issue of this variant is, that it is slow. While the algorithm itself is not very efficient, the main problem here is Selfie's compiler is not performing any optimization. Furthermore, considering Selfie's self-self compilation structures, it can be observed that this variant does not scale at all.

Another issue is the isolation of metadata and memory. As already mentioned, all kind of memory is stored in the same heap. Now consider a buffer overflow caused by a programming error. If the metadata's first entry is overwritten with an invalid address, the collector would crash. Even worse, accidently setting this pointer to zero (i.e. list ends prematurely) or to a previous entry (i.e. creates loop → infinite list search) leads to nondeterministic behavior. The former case means, that actually valid metadata might not be considered by the collector anymore meaning that they and their corresponding objects become essentially uncollected memory.

3.2 Syscall

The syscall variant is an emulator specific implementation. This is implemented by catching a certain system call and executing a different implementation. The concept behind this is moving the collector out of the mutator's into the system's memory space, leading to an isolation of collector and application. From the application's point of view, the collector does not even exist, eliminating the need to distinguish between collected and uncollected memory as all memory is collected. Figure 12 provides a visualization of this concept.

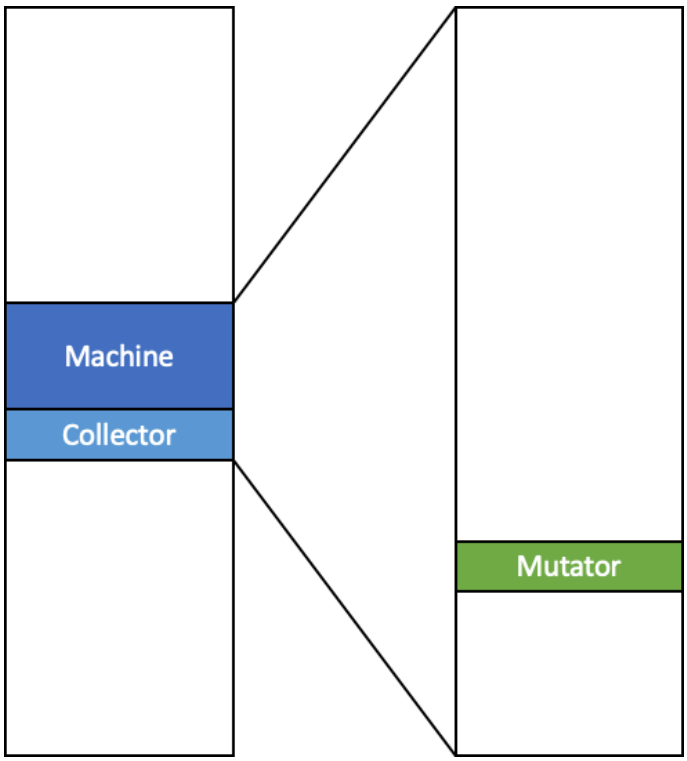

Figure 12: structure of gc syscall

3.2.1 Implementation

Just as mentioned in 3.1.1 Implementation: bound fetcher, metadata management, memory allocation and calling convention need to be considered.

3.2.1.1 Bounds

Bounds can be determined easier when using the syscall variant as opposed to the library variant since they are contained in the machine's context. This context structure is extended to also feature used/free list heads as well as the period counter and "enabled" flag.

3.2.1.2 Memory Allocation

Before explaining the garbage collector's memory allocation, let's take a look at Selfie's bump pointer allocator. Malloc is implemented as follows: Retrieve the size of memory to be allocated and the current bump pointer (stored in data segment), try to set the program break to $bump + size$ using the break (brk) system call, perform validity check (i.e. new program break > old program break or new program break = old program break if size = 0), return result (i.e. old program break on success and a null pointer on failure).

This allocator structure can be taken advantage of, by intercepting the break system call. Changing its implementation to feature garbage collection eliminates the need to recompile binaries to use garbage collection, meaning that all RISC-U binaries can be executed with garbage. To do that, the new implementation needs to conform to the default break's input and output semantics. Figure 13 visualizes what that means to the calling convention of both allocator variants.

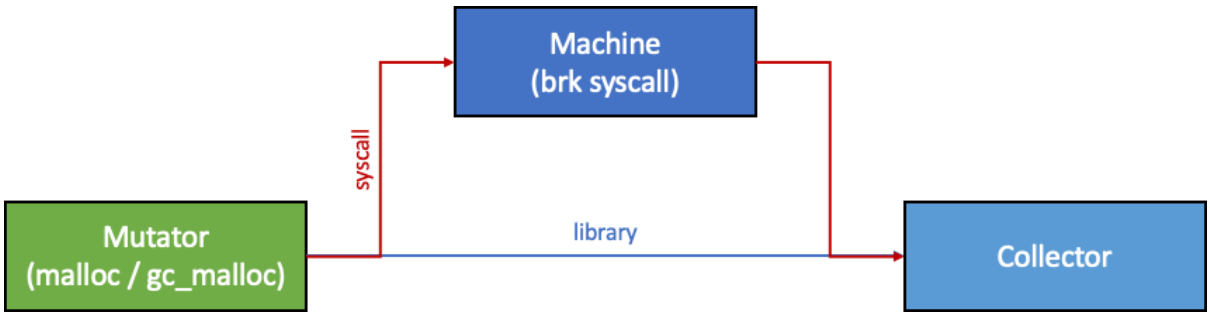

Figure 13: calling convention - syscall vs. library

Input semantics: The break syscall's input parameter is the new program break to be set. All inputs less than or equal to the current program break are considered invalid and redirected to the default implementation. Otherwise the object size is calculated by subtracting the old program break from the new program break. After this translation, the gc_malloc function is called which handles garbage collected allocation.

Output semantics: gc_malloc returns the pointer to the allocated/reused object if successful (otherwise a null pointer, conforming to malloc semantics). However, in case of a reused object this return value is smaller than the current program break. This is where the gc break output semantics differ from the default break output semantics, since default break always returns a value greater than or equal to the current program break. However, since all values less than or equal to the current program break have previously been redirected to the

default break implementation, the gc break system call's semantics remain correct. If such values stem from the gc_malloc implementation, malloc's sanity check needs to be removed as it would change the valid (reused) pointer to zero otherwise (see [6] → implement_gc_brk).

3.2.1.3 Metadata Management

Metadata is still allocated for each created object. However, as opposed to the library version metadata is stored outside of the mutator's address space. This means, that there are different allocators for metadata and object memory. Metadata allocation is simply handled by the system's allocator itself. Object allocation on the other hand uses the break implementation to create memory.

Last but not least, the machine's virtual memory also needs to be accessed (i.e. during mark and sweep). Selfie uses paging to manage and provides functions to access a machine's memory. A potential issue could arise when accessing unmapped memory. There are only two cases in which data is stored to/loaded from the virtual memory: Load during marking and store during reuse (in case an object is zeroed before reusing it). Let's also consider how Selfie allocates virtual memory: On boot level 1 or above all mallocated memory is zeroed [6]. This means, unmapped memory is always 0 since the garbage collector does not work on boot level 0 anyway (see 4.3 Garbage Collection at Boot Level 0). Accessing memory simply checks if the location has been mapped already and if not returns 0 or does not store memory otherwise.

3.2.2 Advantages

The syscall variant is faster and more scalable than the library variant. Instead of emulating the code it is run on boot level 0 and can benefit from initial compiler optimization (since Selfie needs to be compiled using the host machine's C compiler). Furthermore, metadata is isolated from the actual memory, meaning that overflow bugs cannot be caused by the mutator. Binaries are also usable without recompilation.

3.2.3 Disadvantages

Contrary to the library variant, the syscall variant lacks portability. The compiled executables are not garbage collected on any other emulator/machine not supporting this garbage collector flag. Furthermore, it is very Selfie dependent.

3.3 Code Sharing between Library and Syscall

The main issue of supporting two versions of the same garbage collector in one project is code duplication. Therefore, the main goal was to implement it in a way that would require little to no code duplication.

Let's consider the differences of both variants again: bound fetcher, memory allocation, metadata management and calling convention (also different memory access when using the syscall variant). All of these issues apart from calling convention can be solved by introducing wrapper functions. Before implementing those, a way to distinguish between library and syscall during runtime is needed. This is solved by passing a context pointer to all garbage collector functions. If this variable is a null pointer, the library and otherwise the syscall

variant is used. These checks are performed at the lowest level possible (i.e. only inside of wrapper functions), as to not clutter the code.

For example: `uint64_t get_gc_enabled_gc(uint64_t* context) [6]` is a wrapper function to return whether or not the garbage collector is enabled (and initialized). First off, a variant check is performed using the context parameter. In case of the library variant, the value of a global variable is returned (i.e. USE_GC_LIBRARY), otherwise the context is queried.

The same wrapper function structure is also used for list heads, bounds and gcs in period (see 2.4.5 When to Collect). There are wrapper functions for setters too. Functions to allocate metadata and access memory are also implemented in the same way. This wrapper functions make up the majority of code differences, apart from calling convention which has to be done by the mutator.

Last but not least, the initialization of the collector differs between variants too: The syscall variant automatically initializes the collector before starting code execution, while the library has to be manually initialized by the mutator.

4 Garbage Collection in Selfie

As already mentioned, Selfie provides a compiler, emulator and hypervisor. These components can be combined to create different constructs (e.g. self-self compilation). This section observes the interaction between those and the garbage collector.

Let's start by taking a look at Selfie's nomenclature: Mipster and Hypster are the nicknames of Selfie's emulator and hypervisor respectively. Executing Selfie on a host machine is referred to as boot level 0, while all further emulator/hypervisor layers are referred to as boot level 1, 2, etc. Mipster and Hypster can be stacked in arbitrary order with the exception of boot level 0. Boot level 0 can only execute a Mipster (i.e. boot level 1 is always a Mipster).

4.1 Library vs. Syscall

Selfie's prime example is its self-self compilation. Figure 14 shows the difference in structure between both variants. Similarly to Hypster not being available on boot level 0, the library version of the garbage collector is also not available on boot level 0 (see 4.3 Garbage Collection at Boot Level 0).

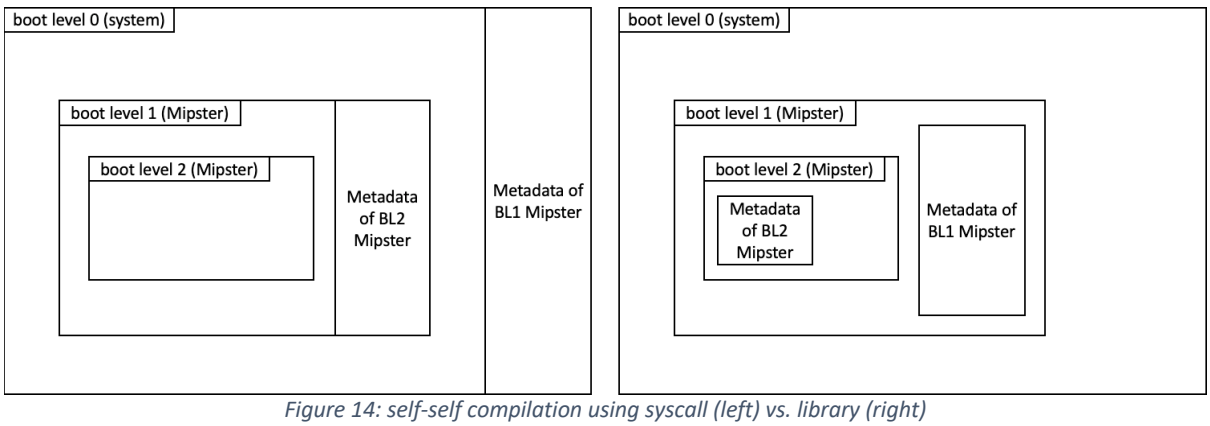

Figure 14: self-self compilation using syscall (left) vs. library (right)

Replacing boot level 2's Mipster with a Hypster not only moves its metadata but also changes execution of the syscall variant. Instead of running the garbage collector code of BL2 on boot level 1, it is run at boot level 0. This results in speedup, since each layer of Mipster increases the instruction count exponentially.

4.2 Self-Collecting Garbage Collection

Suppose a binary containing the collector in its library variant. This binary is executed on a Mipster using the syscall variant. Figure 15 shows this structure in action. When allocating memory, the library's gc_malloc function creates an object and its metadata. Since both are allocated, using the system's allocator (i.e. the syscall variant), two metadata entries are created. This in and of itself does not lead to any collection, since the base setup of the collector uses a free list, meaning all metadata and objects are always reachable. However, suppose this is no longer the case (i.e. perform garbage collection, but do not reuse any memory). The library dropping objects like that means that they can no longer be considered by the collector. This is where the second garbage collector comes into play: It keeps all object in its free list, meaning that the memory can be reused at any time. Therefore, also metadata entries can be reused at any time.

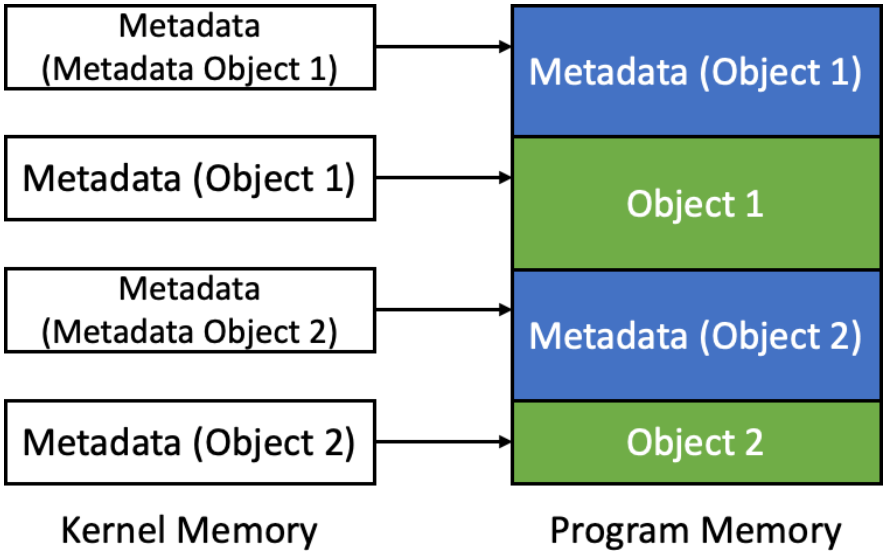

Figure 15: self-collecting collector

4.3 Garbage Collection at Boot Level 0

The implemented collector requires the architecture as defined in 1.4 The Target Architecture. in order to work. Usually Selfie is executed on x86_64 machines running Windows, Mac OS or Linux. Since their kernels and the x86_64 architecture in general are much more complex, garbage collection is not supported at system level. However, this does not mean that it is impossible, since collectors like [10] are supported on a wide range of systems. Furthermore, running the compiled binary on an actual RISC-V machine, enables the collector to work on boot level 0 as well.

5 Analysis of the Algorithm

This section covers the asymptotic space/computational complexity of the algorithm. There are three cases which need to be considered: Collection, reuse and new allocation. Compiler, architecture and optimization issues resulting from the nature of conservative mark/sweep collectors, are not considered. This analysis defines the following variables:

$u \dots number\ of\ used\ list\ entries$
$f \dots number\ of\ free\ list\ entries$
$sizeof(x) \dots function\ to\ determine\ size\ of\ object\ x\ [in\ words]$
$d \dots size\ of\ data\ segment\ [in\ words]$
$s \dots size\ of\ stack\ segment\ (at\ current\ time)\ [in\ words]$

5.1 Collection

Collection can be subdivided into Mark and Sweep phases.

5.1.1 Mark

First of all, let's take a look at Code block 2. Mark segment is executed for both data and stack segment, meaning all words of both segments are traversed. For each of those words, reachability analysis is carried out. This reachability analysis takes $O(u)$, since a list search on the used list is performed. If the reachability analysis succeeds, the mark object function (Code block 3) is called recursively. Without considering the recursive marking, the asymptotic computational complexity of mark_segment is $O((d+s)*u)$.

Looking at Code block 3, it can be observed that each object can be marked at most once leading to a maximum of $h = \sum_{(i=1)}^{u} sizeof(object_i)$ words (i.e. each used object is reachable). In combination with the complexity of the list search, the upper runtime bound of all recursive mark_object calls is $O(h*u)$.

In total the asymptotic complexity of Code block 2 and Code block 3 combined is $O((d+h+s)*u)$. However, this is only the theoretical upper bound. There is also a lower bound: Before conducting a list search (reachability analysis) a basic sanity check is done. This sanity check has a runtime of $\Theta(1)$. Suppose this sanity check fails for each of the root segment's words, meaning that no list search is performed at all. Mark would still need to traverse both data and stack segments leading to an asymptotic computational bound of $\Omega(d+s)$. Furthermore, an optimized reachability analysis would improve the upper bound to $O((d+h+s)*\log u)$ (using a tree instead of a list) or even $O(d+h+s)$. For further information about possible improvements see: 7 Optimization.

5.1.2 Sweep

Sweep (Code block) has an asymptotic complexity of $\Theta(u)$, since each used list entry is checked and if list removal is carried out, its runtime is $\Theta(1)$.

5.1.3 Summary

The total asymptotic complexity of the collect algorithm is $O((d+h+s)*u)$.

5.2 Reusing Memory

Collected memory might be reused. To do that a list keeping track of all free memory of the memory pool (Figure 10) is required. Searching this list takes $O(f)$. Usually, the memory of a reused object is zeroed before returning the object. This emulates Selfie's behavior of zeroing all allocated memory on boot level 1 and above. This leads to a total of $O(f + sizeof(object))$. Alternatively, object zeroing could be disabled to improve performance. However, this would lead to a violation of Selfie's zalloc semantics (i.e. Selfie emits malloc and zalloc as the same function [6] during compilation). Reusing memory without zeroing could lead to memory containing nonzero words. There is also an additional flag which disables the reuse of memory altogether.

5.3 Allocating New Memory

If reusing is turned off or there is simply no memory to reuse, new memory needs to be allocated. That includes allocating both metadata and object memory, setting initial metadata values and pushing the new entry onto the used list. All of that can be done in $O(1)$. If reusage of memory is turned off and the collector is not invoked in an instance of gc_malloc, the runtime of gc_malloc is $O(1)$. Otherwise if reusing is turned on, the runtime is $O(f)$ if the collector is not invoked.

5.4 Relation between Time and Space Complexity

Not each gc_malloc call might lead to a collector invocation. Collection is only performed if a certain criterion is satisfied (see 2.4.5 When to Collect). This criterion is not ideal and there are much more efficient ways to determine whether to collect or not (e.g. memory pressure). However, it provides a simple way to analyze how collection influences both runtime and space complexity.

Speaking about space complexity, up until now only asymptotic computational complexity has been discussed. Asymptotic space complexity is influenced by its period constant which can be modified and optimized to suit a certain application. Keep in mind: As soon as the period constant is set to a value larger than 0, the optimal space complexity achievable by this algorithm is no longer guaranteed.

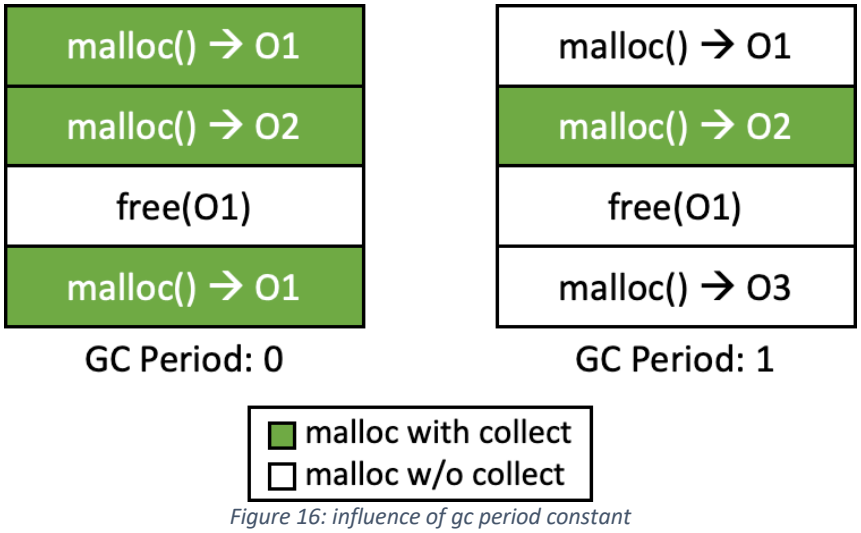

Figure 16: influence of gc period constant

This can be shown using a simple example (Figure 16, right side): Consider a period of 1. Now let's allocate three objects. Allocating the first one means no collection, which is fine since there are no objects in use anyway. Allocating the second means that collect is invoked (which doesn't collect anything as there is no unreferenced object). Now let's free object 1 and allocate another one. Using a gc period of one leads to new allocation, since collection is skipped in uneven allocations (i.e. period counter is 0). This is not optimal considering space complexity, since object one is unreachable and therefore free and reusable. Examples like that can be constructed for every gc period greater than 0.

Another space complexity issue is caused by the modified first fit strategy (see 2.5.2 Reusing Free Memory and 7.3 Reusing, Fitting and Free List Improvements).

6 Experiments

The experiment setup is as follow:

- OS: Arch Linux
- Processor: Intel Core i7-4900MQ @ 2.8GHz
- RAM: 32 GiB
- Compiler: GCC 10.1.0 (-O3, -m64)

The median of five runs is used in all graphs.

The first experiment looks at the runtime of not using a garbage collector vs. using a garbage collector on Mipster and Hypster. This means that Selfie is compiled, executed on Mipster to compile itself again. The result is subsequently executed on another Mipster/Hypster layer. Note: This experiment uses a gc period constant of 0 to show just how much collecting during each allocation affects performance. The performance difference between Hypster and Mipster can be explained as follows: Without parameters, selfie simply prints a synopsis, meaning not that much code is actually executed. Still, most of the executed code is actually the garbage collector and Figure 17 shows that there is quite a difference between Mipster and Hypster considering this basic execution. However, when looking at the memory consumption of not using a garbage collector vs. using one, it can be observed that the collector uses a lot less memory. Obviously both Mipster and Hypster variants use the same amount of memory in when using garbage collection.

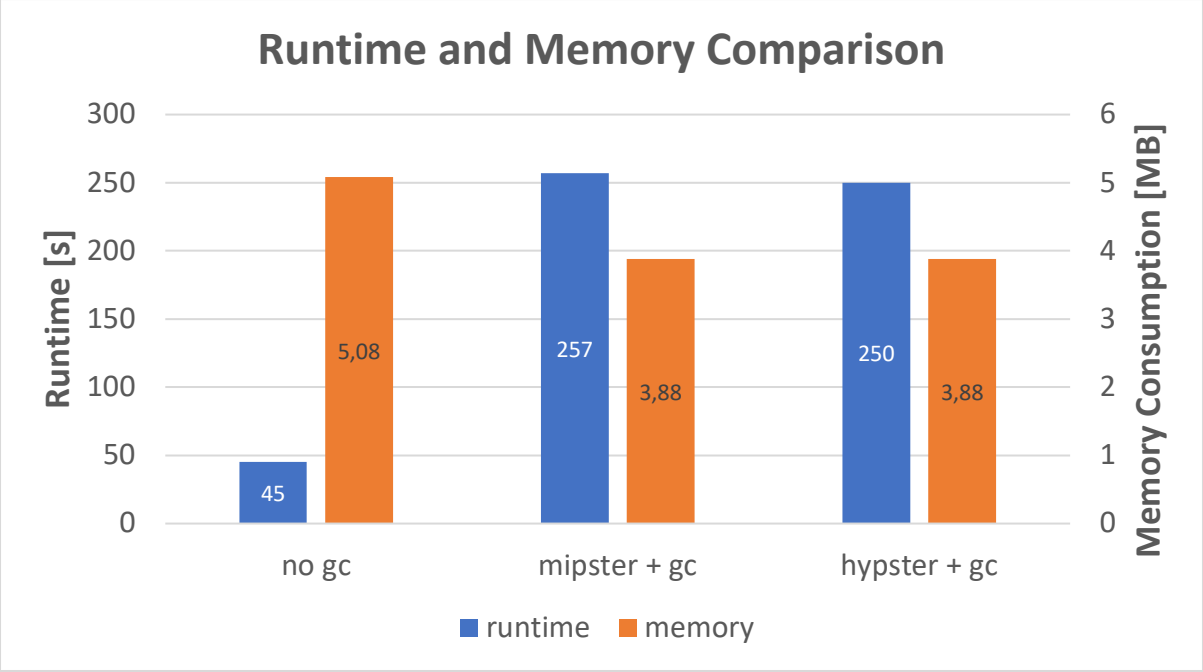

Figure 17: comparison of using no gc vs. using gc with mipster and hypster

Next up the runtime of the library and syscall variants have been compared. The same Mipster on Mipster construct has been used for this experiment. A gc period constant of 12000 has been used in order to reduce runtime. This collector coefficient has the same effect on both library and syscall version. Looking at Figure 18 it can be observed, that the library is about 6 times slower than the syscall variant. As already mentioned, there are two main reasons, that the syscall variant is much faster than the library variant:

1. Collector is executed on a lower level (→ less overhead)
2. The collector of the system level benefits from host compiler optimization

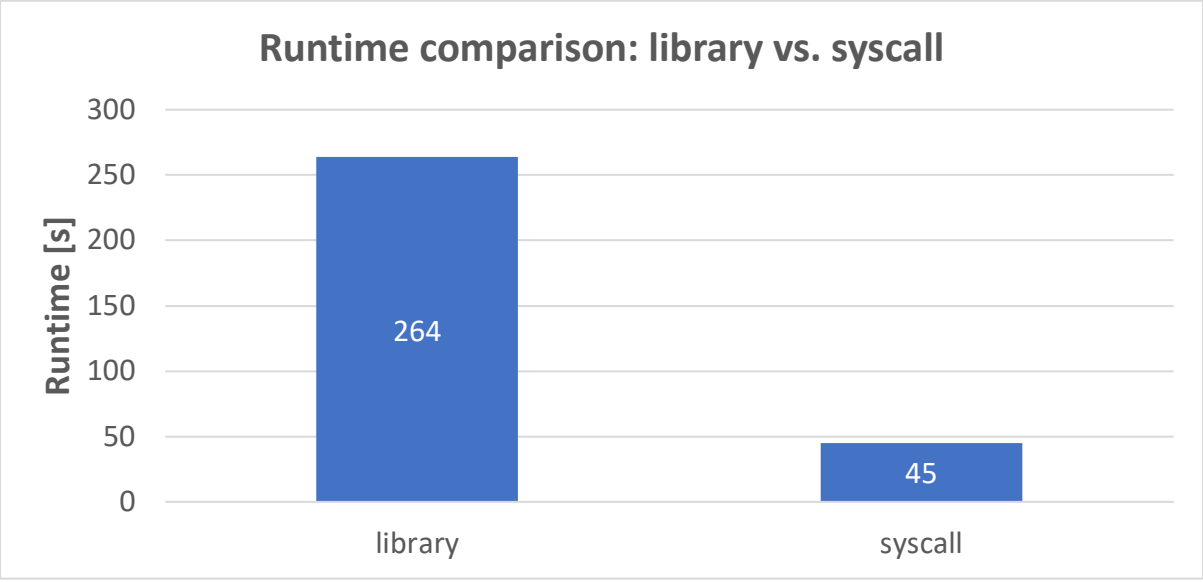

Figure 18: runtime comparison (library vs. syscall)

Figure 19 compares the time/space complexity of different gc period coefficients. Keep in mind, that the coefficient leads to different results in different applications. The aforementioned Mipster/Mipster structure has been used for this experiment as well. The sweet spot is somewhere between a coefficient of 10 and 100.

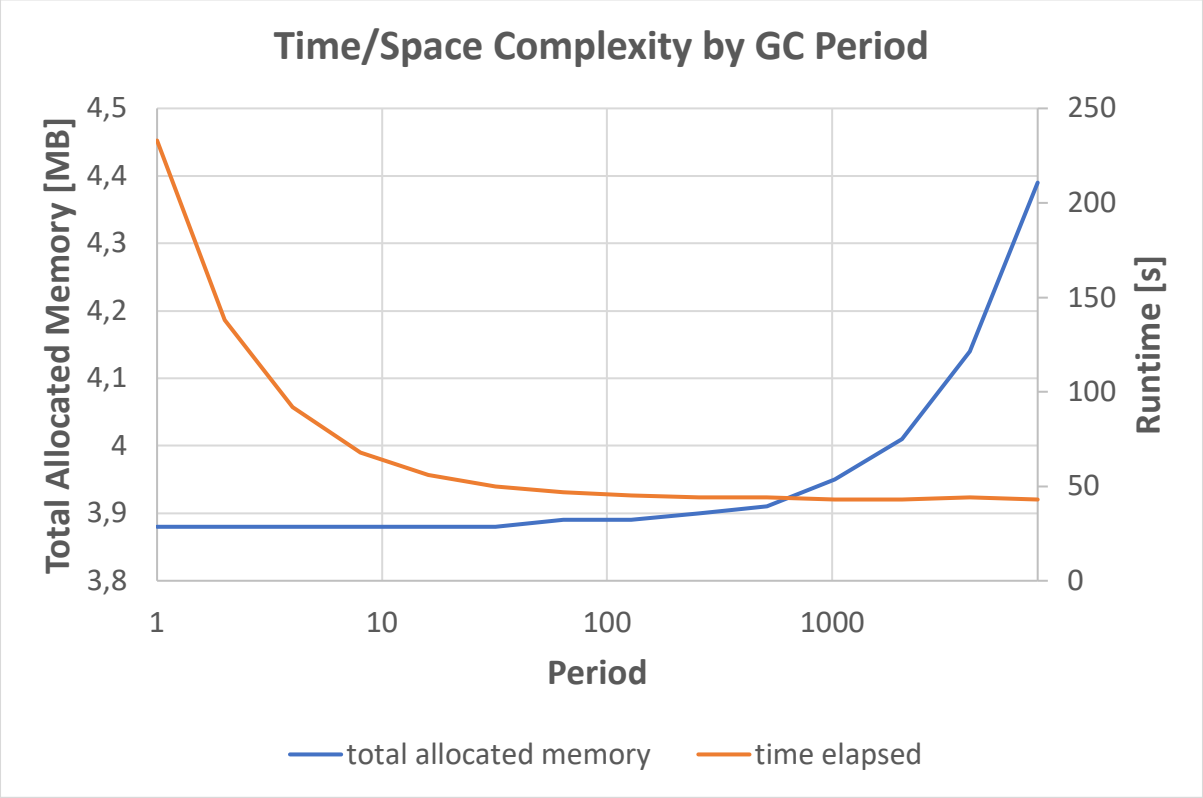

Figure 19: comparison - runtime/space complexity

Apart from total memory consumption, the relation between metadata and object memory is also noteworthy. A higher percentage of metadata memory typically means, that there is a larger number of smaller objects. Thus this increase indicates that there are way more objects when collecting less frequently (Figure 20 and Figure 21). The reason why that matters is Selfie using a lot of small, short-lived (symbol table entries for compilation) and only a few large objects. Other applications may use fewer, but larger objects leading to a way worse memory performance. Again, the gc period coefficient must be fine-tuned for each application.

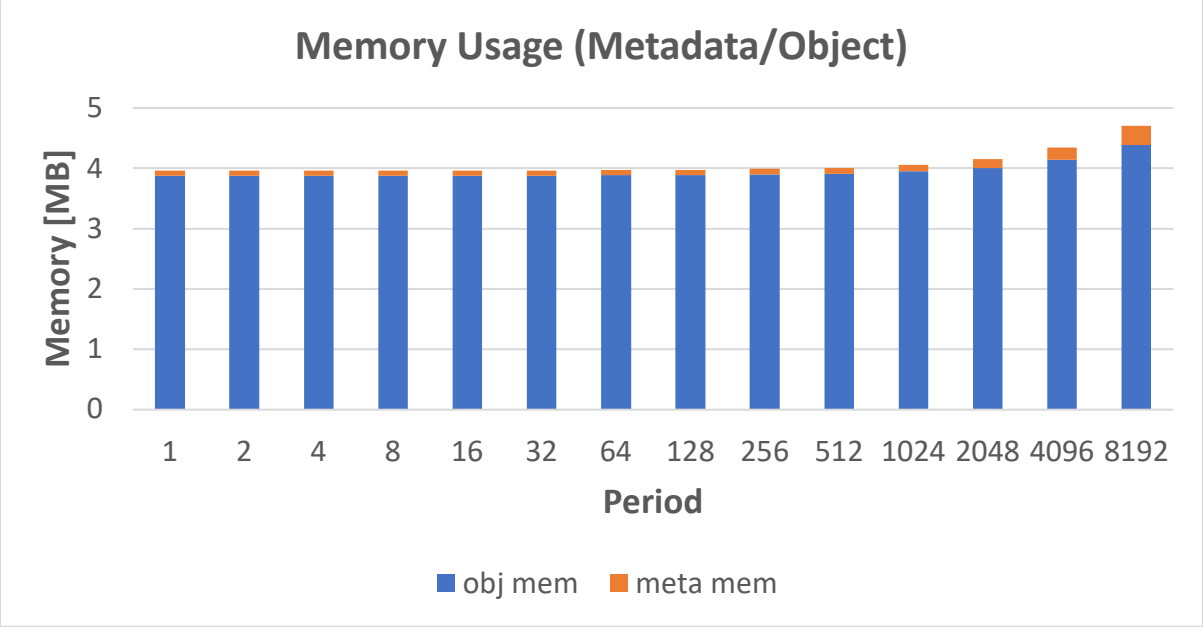

Figure 20: memory usage by metadata/object

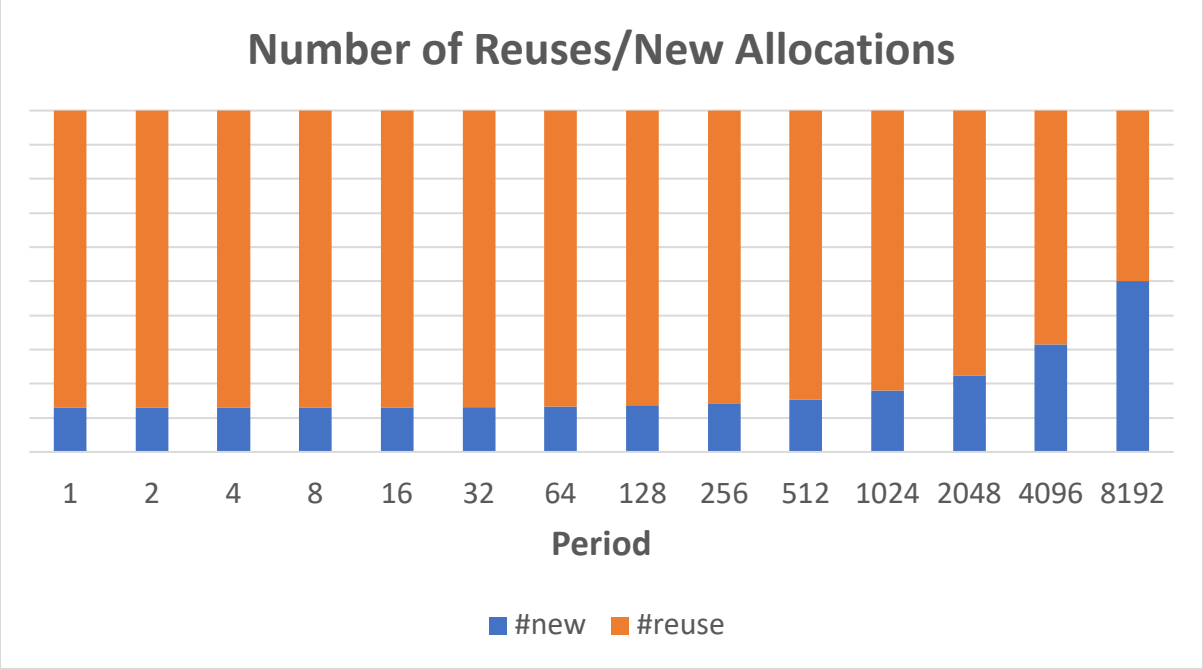

Figure 21: reuse/new allocations

7 Optimization

A garbage collector can use up a lot of runtime. While it is not the aim of this thesis to compare different optimization techniques, there is some basic optimization which can be implemented without making the collector design too complex. This section covers optimization issues of the collector and suggests ways to improve those.

7.1 When to Collect

This is the first obvious question to ask in any garbage collector implementation. As already mentioned, the algorithm uses a simple period criterion. The main advantage of this approach is deterministic behavior. This is especially useful when comparing the garbage collector's runtime and space complexity (see 6 Experiments). The main issue of this approach is the dependence on the application. Just because a coefficient yields good results for one application doesn't mean that this coefficient will yield good results for another one. The reason behind this, is the criterion not using any information about the memory itself. One simple solution would be using heap information in combination with a threshold value (i.e. $heap\_usage = \frac{allocated\_memory}{total\_memor}$; collect only if heap_usage exceeds threshold). Another solution would be checking if the allocation failed and only collect if it did. More advanced garbage collectors like the JVM's collectors use different ways to determine when and especially how long the collector can be executed [11].

7.2 Improving Reachability Analysis

The implementation's reachability has an asymptotic computational complexity of $O(u)$ with u being the number of used objects. This is obviously not ideal, since other mark-sweep collectors can perform it in constant time. While using specialized structures like [3] was not an option for this project, there are still ways to improve performance by replacing the list with a more suitable data structure. One example would be an AVL tree, which has a search complexity of $O(\log n)$ [12]. Even better, a hash map pushes the complexity down to $O(1)$. Therefore, the total runtime complexity of the collect phase could be reduced to the number of words contained in data segment, call stack and heap segment as mentioned by [13].

7.3 Reusing, Fitting and Free List Improvements

Using a different fitting strategy would not only improve space complexity, but also time complexity (when reusing memory). Just as with reachability analysis there are more intelligent data structures to handle free objects, but the algorithm itself could be changed in a minimally invasive way (i.e. changing the list to a more suitable structure) to improve performance by a margin. As an example, consider a tree. This tree is sorted ascending by object size. The lookup criterion changes from "matching" to "smallest possible space in which object would fit" (i.e. best fit).

Another way would be to introduce a small object free list similar to Boehm [3]. The idea behind it being the fact that smaller objects are usually more common than larger objects. Using a free list for each object size (up until a given threshold), leads to a lookup time of $O(1)$. Larger objects would have to be handled differently though.

Most of these suggestions only cover runtime and not space complexity though. If space complexity needed to be improved, a better fitting strategy like best fit or compact fit [14] would have to be employed.

8 Conclusion

Time to review; Before implementing this collector, Selfie had no way of freeing memory. Once allocated, the memory was in use until Selfie terminated. While not a problem for typical execution, it is obvious that always allocating new memory does not scale. Furthermore, Selfie actually uses a lot of small, short-lived objects which make it the perfect candidate for garbage collection. This project addresses a lot of memory allocation problems Selfie had. The garbage collector not only works in the same space as the mutator, but also in a different address space meaning it makes the collection possible, without even running inside of its address space. All of which while not really changing the algorithm itself. Furthermore, the collector can even use both versions of deployment at the same time meaning that the collector can be used to collect its own garbage. Additionally, the collector's parameters can be tweaked to improve runtime or space complexity. Since the collector has been kept as simple as possible, it shows weaknesses in terms of runtime, however there are still ways to improve this design without sacrificing too much of its simplicity.

9 Bibliography

[1] Oracle, "Introduction to Garbage Collection Tuning," 2014. [Online]. Available: https://docs.oracle.com/javase/9/gctuning/introduction-garbage-collection-tuning.htm. [Accessed 21 September 2020].

[2] Oracle, "JVM Specification," [Online]. Available: https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-2.html#jvms-2.5.3. [Accessed 21 September 2020].

[3] M. W. Hans-Juergen Boehm, "Garbage Collection in an Uncooperative Environment," 1988.

[4] Microsoft, "Microsoft Docs Archive (blog of abhinaba)," Microsoft, 27 January 2009. [Online]. Available: https://docs.microsoft.com/en-gb/archive/blogs/abhinaba/back-to-basics-reference-counting-garbage-collection. [Accessed 22 September 2020].

[5] S. D. G. G. Salagnac, "Fast Escape Analysis for Region-based Memory Management," Sciencedirect.com, 2005.

[6] Selfie Project Authors, "Selfie Github," [Online]. Available: https://github.com/cksystemsteaching/selfie. [Accessed 20 September 2020].

[7] RISC-V Foundation, "RISC-V ELF psABI specification," [Online]. Available: https://github.com/riscv/riscv-elf-psabi-doc. [Accessed 20 September 2020].

[8] University of California (Regents), "RISC-V PK Github," 2013. [Online]. Available: https://github.com/riscv/riscv-pk. [Accessed 20 September 2020].

[9] User:M17, "Tracing Garbage Collection Wikipedia article," 24 October 2012. [Online]. Available: https://en.wikipedia.org/wiki/Tracing_garbage_collection#/media/File:Animation_of_the_Naive_Mark_and_Sweep_Garbage_Collector_Algorithm.gif. [Accessed 20 September 2020].

[10] H.-J. Boehm, "A garbage collector for C and C++," [Online]. Available: https://hboehm.info/gc/. [Accessed 21 September 2020].

[11] Oracle, "JVM Garbage Collection Tuning Guide," Oracle, 1993. [Online]. Available: https://docs.oracle.com/javase/8/docs/technotes/guides/vm/gctuning/ergonomics.html. [Accessed 21 September 2020].

[12] E. Alexander, "CS367 Notes: AVL-Trees," 2011. [Online]. Available: http://pages.cs.wisc.edu/~ealexand/cs367/NOTES/AVL-Trees/index.html. [Accessed 21 September 2020].

[13] H.-J. Boehm, "Mark-sweep vs. copying collection and asymptotic complexity (page describing IWMM'95 presentation)," [Online]. Available: https://www.hboehm.info/gc/complexity.html. [Accessed 21 September 2020].

[14] H. Payer, "The Tiptoe Project: Real-Time Operating System Research - Compact Fit," 19 September 2007. [Online]. Available: http://tiptoe.cs.uni-salzburg.at/compact-fit/index.html. [Accessed 21 September 2020].