Self-collecting Goroutines: Short-term Memory Management in Go

MASTER’S THESIS

to obtain the Master’s degree
at the Faculty of Natural Sciences of the
University of Salzburg

submitted by
Dipl.-Ing. (FH) Michael Lippautz

Academic supervisor
Univ.-Prof. Dr.-Ing. Christoph Kirsch

Department of
Computer Sciences

Salzburg, October 2011
Some people never go crazy. What truly horrible lives they must live.

Charles Bukowski (1920 – 1994)

First of all, I want to thank all people who made this work possible.

I would like to express special gratitude to Prof. Christoph Kirsch for his excellent supervision and for providing a very professional and productive working environment. Without his support and suggestions probably none of the following ideas would have been engineered to this extent.

I am also very thankful for the warm welcome that has been given to me by Computational Systems Group. Working with you all is not only a lot of fun, but also very productive. You all promote one’s creativity and try to assist as much as possible.

Thanks (once again) to my family for always standing behind me and supporting my decisions, whether they turn out to be good or bad.

I also want to thank my girlfriend Eva for providing me with all the moral support in this busy time. Thank you for being so patient and also for listening to things you are (mostly) not interested in. Everything seems to be so much easier when you are around. Thank you for that!

One final acknowledgement goes to all my friends who always try to bring some change into my life. Whether it is by contributing to my studies or by keeping me entertained with funny evenings. The time always went by quickly, maybe sometimes too quickly.
Abstract

Dynamic heap management is typically done either explicitly using malloc and free calls or implicitly using a garbage collector. Both approaches usually rely on persistent memory, i.e., memory is allocated for an unknown amount of time. A more recent memory management model, so-called short-term memory (STM), uses expiration dates to limit the lifetime of objects. This model can be implemented in explicit or implicit fashions.

We have designed and implemented an explicit memory management system based on STM in Go. The backwards-compatible implementation uses so-called self-collecting mutators (SCM) that utilize in-band collection algorithms of objects to avoid additional threads. The system does not introduce any locks and is therefore scalable in multi-threaded environments. Furthermore, it is designed to only add constant-time overheads making it incremental. The explicit implementation trades off correctness of a garbage collected system against speed and the ease-of-use of STM. Experiments confirm the proposed model and implementation and demonstrate competitive performance in throughput, latency, and memory consumption compared to the original Go system.
## Contents

Acknowledgements .................................................................................. i  
Abstract .................................................................................................. ii  

1. Introduction 1  
   1.1. Outline of the Thesis ................................................................. 2  
   1.2. Contributions ........................................................................... 3  

2. Heap Management - A Selective Introduction 4  
   2.1. Basics ..................................................................................... 4  
   2.2. Properties ................................................................................ 8  
       2.2.1. Time .................................................................................. 9  
       2.2.2. Space ............................................................................... 10  
   2.3. Explicit Memory Management .................................................. 12  
   2.4. Implicit Memory Management ................................................... 13  
       2.4.1. Why? - Benefits and Drawbacks over Explicit Memory Management 14  
       2.4.2. Garbage Collector Types ..................................................... 15  
       2.4.3. Tracing Garbage Collection Characteristics ............................. 18  
       2.4.4. Naïve Mark-Sweep Garbage Collection – An Example ............... 20  

3. Short-term Memory (STM) 23  
   3.1. Expiration of Objects ............................................................... 23  
       3.1.1. Liveness - short-term memory (STM) vs. Traditional Malloc/Free . 25  
   3.2. Model ....................................................................................... 26  
       3.2.1. Single- and Multi-Refreshing ............................................... 27  
       3.2.2. Global Time compared to Go Channels .................................. 28
4. Short-term Memory Implementation in Go

4.1. The Go Programming Language - A Technical Examination
   4.1.1. Go Specific Concepts
   4.1.2. Heap Management in Go
   4.1.3. Building and Deployment

4.2. Go Runtime
   4.2.1. Allocator
   4.2.2. Garbage Collector
   4.2.3. Slices

4.3. Hybrid Self-collecting Mutators
   4.3.1. Application Programmable Interface
   4.3.2. Self-collecting Mutators – Data Structures

4.4. Runtime Modifications and Compatibility
   4.4.1. Memory Layout
   4.4.2. Hybrid Operation

4.5. Implementation and Complexity
   4.5.1. Allocation Modifications – malloc
   4.5.2. Allocation Modifications – mcentral
   4.5.3. Allocator Modifications – scm_free
   4.5.4. StmTick
   4.5.5. StmRefresh

4.6. Complexity Summary

4.7. Example

5. Experiments

5.1. Test Environment

5.2. Single-Threaded – Tree Benchmark
   5.2.1. Results

5.3. Multi-Threaded – A Webserver
   5.3.1. Results

6. Conclusion

6.1. Usage – Well Suited Applications

6.2. Future Work
Contents

Bibliography 73

A. Abbreviations 76

B. Computer language benchmarks game – tree benchmark 77

C. Webserver benchmark 80
List of Figures

2.1. Unix process memory layout [1] .............................................. 5
2.2. Object graph ................................................................. 7
2.3. External fragmentation ..................................................... 11
2.4. Internal fragmentation ...................................................... 11
2.5. Precise and conservative reference detection ......................... 19
2.6. Naïve mark-sweep .............................................................. 21

3.2. Allocation for one time unit [2] ........................................... 24
3.3. Allocation of an object for seven time units using further refreshing [2] ... 25
3.4. Different sets of objects on the heap [3] ................................. 26
3.5. Sets of objects over program time ...................................... 27

4.1. Go memory layout and contention levels ................................ 37
4.2. Memory and bookkeeping structures relations ......................... 38
4.3. Internal memory layout of a slice ....................................... 39
4.4. SCM data structures .......................................................... 41
4.5. Object header for “small” and “large” objects ......................... 42
4.6. Modified memory layout of MSpan ....................................... 45

5.1. Tree benchmark overall execution time ................................ 64
5.2. Tree benchmark overall live heap size ................................ 66
5.3. Tree benchmark latency measurements ................................ 67
5.4. Webserver live heap-size ................................................... 68
5.5. Webserver latency measurements ........................................ 69
# List of Tables


4.1. Summary of function/modification complexities .............................. 58

5.1. System configuration ...................................................................... 61
5.2. Overall time [s] – tree benchmark .................................................. 65
5.3. Tree benchmark latency summary .................................................... 67
5.4. Webserver benchmark latency summary .......................................... 70
Listings

2.1. Allocator interface ............................................. 12
2.2. Cyclic memory leak algorithm in Go .............................. 16

4.2. Go sending and receiving [5] ...................................... 33
4.3. Go SCM API [2] .................................................. 40
4.4. mallocgc modifications ........................................... 47
4.5. MCentral_Grow modifications .................................... 48
4.6. Schematic scm_free ............................................... 50
4.7. Schematic StmTick ............................................... 51
4.8. Schematic Expire_Local ......................................... 52
4.9. Schematic Expire_If_Exists ..................................... 53
4.10. Decrement-and-test \(\rightarrow\) exchange-and-add ................. 53
4.11. \texttt{atomic\_int32\_exchange\_add} ................................ 53
4.12. StmRefresh and internal_refresh ............................... 55
4.13. Go STM example ................................................ 58

5.1. Go tree benchmark transformation .................................. 63

C.1. Webserver using the Go garbage collector (GC) .................. 80
C.2. Webserver using STM and the GC ................................ 80
1. Introduction

One of the key aspects of computer programs, that need a variable number of objects, is how the memory to store these objects is allocated, and how long it should be available. Systems that try to solve these kind of problems and provide an interface to control the flow of objects are called memory management systems. These systems provide a way to create new objects using an allocation method, and rely on either explicit or implicit approaches to get rid of previously allocated objects. In the explicit way, the programmer is in charge of freeing the memory himself, while in the implicit approach a system, the so-called garbage collector, is responsible for removing unreachable objects. Both approaches suffer from design-related drawbacks. The explicit approach, on the one hand, is fast but prone to errors such as dangling pointers or double frees. Implicit systems on the other hand guarantee safety regarding certain types of errors, but may impose a significant management overhead. The key aspect that both of these types of memory management systems have in common is that they deal with objects that are not needed anymore.

In order to provide a different point of view on programs and programming habits we utilize a system called short-term memory (STM), which allows the programmer or utilizing system (i.e. a garbage collector) to keep the focus on still needed objects and just forget the other ones. Despite giving an introduction on the general model we also show the general programmable interface that is used to control the flow of objects in a STM enabled program.

Furthermore, we present an implementation approach of the proposed STM model that relies on so-called self-collecting mutators (SCM). SCM provide a fully incremental collection strategy that does not need to be separated into a different thread, but can run in the actual mutator (user program), hence self-collecting mutators.

We then present an implementation of the SCM that can be used from within the Go programming language environment. The Go language, introduced in 2008 by Google, is a language that aims to be fully typed (although without any hierarchy), concurrent
and garbage-collected. Go is implemented as compiled language with all programs being linked against a runtime library. Short-term memory is integrated in the runtime library to provide a fully compatible approach where objects can either be managed by the existing system (compatible mode) or be converted into short-term objects. While this makes the language unsafe in principle, it enhances the performance. We also claim that it is easy to create programs that are safe, although not following a proof.

Finally, we present benchmarks comparing the existing baseline system of Go to STM using SCM. The benchmarks are categorized into a performance only tree building benchmark and a multi-threaded webserver serving static html pages. Both benchmarks cover time and space dimensions of the tested systems.

1.1. Outline of the Thesis

We start with an introduction on memory management in general. This introduction is followed by a description of the STM management model in general. After the implementation is introduced, the benchmarks are presented and the thesis is summarized once more in a conclusion.

Chapter 1, Introduction: The introduction explains the motivation and gives an outline of the thesis.

Chapter 2, Heap Management - A Selective Introduction: This chapter introduces the reader to the basics of heap management, including general definitions, as well as categorization into explicit and implicit systems. The underlying principles as well as examples are given for each type of system.

Chapter 3, Short-term Memory: The third chapter deals with STM as general concept. Therefore, these sections deal with the liveness of objects and the general programming model that is used for STM.

Chapter 4, Short-term Memory Implementation in Go: This part then covers the integration of the concept of STM into the programming language Go. Therefore, a backwards-compatible approach has been implemented into the Go runtime libraries. Furthermore, the implementation is also evaluated regarding its complexity/scalability.

Chapter 5, Experiments: The fifth chapter illustrates the results of several performance benchmarks comparing the existing garbage collection system that is implemented in Go with the STM-based approach.
Chapter 6, Conclusion: The last chapter concludes the thesis and gives an insight in what might be the future of memory management using STM.

1.2. Contributions

The main contribution of this thesis is the design, implementation and evaluation of the STM model in Go.
2. Heap Management - A Selective Introduction

This chapter starts with a general overview of dynamic memory management, also called heap management. Therefore basic terms are introduced. Following these definitions the fundamental properties, such as fragmentation, are explained. The chapter then continues with the differentiation between explicit and implicit heap management systems. Both systems are covered by descriptions and examples.

2.1. Basics

One of the most important assets of computer programs are variables that are used to store information. There exist different kinds of places where information can be stored when variables are used in a program. Literature, i.e. Tanenbaum [1], suggest a process memory layout as illustrated in Figure 2.1. This layout is also used by operating systems, such as Linux [6].

Memory allocation can be divided into three different types, that is static memory allocation, stack memory allocation and dynamic memory allocation. Although we deal with dynamic memory allocation, the other two are also described in short, to illustrate the differences.

All three types of memory management use different sections in the memory layout to store their data. The properties that need to be known to distinguish them are the size of the chunk of memory that is needed, and if the time the object will be needed is known at compile-time, or runtime.

**Compile-time vs runtime.** Compile-time is the time the compiler translates the source code into binary code. Runtime, however, is the time when the program is executed.
2. Heap Management - A Selective Introduction

Static memory allocation. Memory is allocated by the compiler. The lifetime and size are known at compile-time.

The sections that hold statically allocated memory are usually data, bss, and stack. The data sections is used for static and global variables that are already initialized. The bss section holds variables that are global or static but have not yet been initialized. The stack space is organized as last-in first-out (LIFO) queue, typically growing down, holding statically or stack allocated variables.

Stack memory allocation. Memory is allocated at runtime in a LIFO manner. The lifetime is already fixed at compile-time, while the amount of memory can be unknown.

Stack allocated memory is stored in the stack segment. The lifetime has to be known at compile-time, since the stack needs to be cleaned up before any returning from an executed function [1].

Dynamic memory allocation. Memory is allocated at runtime and the amount of memory is not known at compile-time.

All dynamically allocated variables are located in the heap section. Theoretically, the heap section is one big space of memory, that could be utilized directly by a program for reading and writing values of variables. However, for practical reasons, i.e. efficiency, the heap is organized by a memory management system.

Figure 2.1.: Unix process memory layout [1]
As the memory is acquired dynamically, meaning at runtime, in a program, every memory management system usually provides two actions that can be called either explicitly or implicitly:

**Allocation [7]**. When a program needs memory it requests it from the underlying memory management system by using an allocation operation.

**Deallocation [7]**. After some time, the previously acquired memory may not be needed anymore. A memory management (MM) system may provide an explicit operating to give this junk of memory back. Deallocation is also called freeing.

Memory that is dynamically acquired using allocation but is never freed anymore is also called immortal memory.

Many programming languages prohibit working on raw memory directly. Instead, they force the programmer to use objects. In this case the allocation calls provide the programmer with an object, instead of a chunk of memory.

**Object [7]**. An object, or also called cell, is a contiguous block of memory (chunk) defined by a logical structure. An object is subject of allocation and deallocation requests.

For instance, an object could contain numbers, strings, or even other objects. The programmer is then obligated to respect this structure and not allowed to decimal values into a string (as numbers). The logical structure that describes an object is called object type.

**Object type**. The object type describes the logical structure of objects. An object is always assigned exactly one object type, but an object type can be assigned to multiple objects.

When acquiring objects or raw chunks of memory, the allocation call does not return the object itself, but a pointer to it.

**Pointer [7]**. A pointer represents a reference to, either an object, or a memory location. Pointers may have type, that is an object type for languages that use objects, or a logical description for other languages. A pointer having a type, also called typed-pointer, references an object or chunk of memory that satisfies the underlying description.

Note that there exists another term called reference that is used in literature to identify a link between two objects. Although there do exist programming languages, such as C++,
where pointers and references do not have the same semantics [8], in this thesis the terms pointer and reference are used interchangeably.

In order to describe certain program states regarding memory management it is important to know how objects are connected to each other via the use of references.

**Object graph [7].** A graph is a set of nodes together with a set of edges connecting them. An object graph is then a graph where objects are nodes and references are directed edges connecting them.

Figure 2.2 shows an object graph where the root set of nodes is visualized by rectangular nodes, while objects not relying on the root set are marked as circles. Each object graph contains entry points, that are referenced from within a so-called root set. The root set contains global variables, stack variables, or registers. While global variables hold permanent data, stack variables hold data with a limited scope and are thus temporary. Whenever an object graph is built, it traverses from the root set, through the objects until no more objects can be reached from existing ones in the tree. In Figure 2.2 the entry points are formed by the objects $A$ and $G$.

In order to reason about objects in the object graph, we need to introduce the term **reachability**.
Reachability [7]. An object is reachable if it is either referred to by the root set, or another reachable variable.

As a consequence of the above definition, all objects that are reachable by a sequence of references from within the root set, are reachable. In the example from above, Object D is reachable, because there exists a sequence A → B → D and A is referenced from the root set.

Finally, one can reason about objects being active, or also called live.

Live [7]. Memory, or an object, is live (active) if the program still needs it, i.e. will read from (or write to) it in the future.

Because it is generally not possible for a program to determine whether an object is live at a given point in time, it is necessary to approximate this liveness. Memory management system like GCs (see Section 2.4) use an object graph and a reachability analysis to check whether an object is live or not, also called dead. Referring once again to Figure 2.2, object D would be live, because there exists a sequence of references to it (A → B → D and A), while object E would be considered as dead, as there exists no sequence of references from the root set to it.

Before describing explicit (Section 2.3) and implicit (Section 2.4) memory management systems, it is important to define their properties, and thus making these two systems comparable.

### 2.2. Properties

In order to reason about memory management systems it is important to have properties that quantify certain capabilities of these systems. While there do exists properties, such as ease-of-use, that are hard to measure, one can also take time and space into consideration. These properties can be measured and create a two-dimensional space where one can position a system depending on its performance in these dimensions.

*Note:* Although it is hard to measure the ease-of-use of a system, it may be equivalently sustainable when comparing different memory management systems.
2. Heap Management - A Selective Introduction

2.2.1. Time

Time properties all deal with the amount of time several parts of the system need in order to be computed. While this is also interesting for a mutator, it is especially important to know how much time a memory management system has consumed.

**Total runtime.** *The total runtime of a program is the time span from the start of the program until it terminates.*

It is important to note that the total runtime also includes time the program may wait for user-input, or resides in the memory manager. Since it includes user interactions, the total runtime may be more important for pure calculations.

**Pause time.** *Pause time is the time a program resides in the memory management system. The pause time may be accumulated to an overall pause time that can then be related to the total runtime.*

While the overall pause time is an indicator of how much time the mutator needs to wait for the memory manager, it is also important to measure the “jitter” for these wait times. This property is called latency. The latency of a program can be calculated by acquiring the maximum pause time of the memory management system. This may either be related to allocation or deallocation, or a combination of both. Furthermore, if these times are hard to measure, it is also possible to measure the exact times at which an allocator operation happens in a loop. The loop, however, has to consist of constant-time and allocator operations only. Otherwise the latency could also originate from other calls than the allocator.

Despite the general performance of a mutator including its allocation system it is also important to further investigate the operations of the memory manager.

**Allocation time.** *The time the allocator needs to acquire the requested amount of memory (or object of a specific size).*

**Deallocation time.** *The time an allocator needs to deallocate memory, i.e. make it reusable through the system, is called deallocation time.*

Both operations may be dependent on the size of the object (i.e. chunk of memory) that is dealt with. They may even be further influenced by the size of the heap, or even previously called operations. In the case that an action is only dependent on the size of the object, or can be calculated in constant time, that operation is called predictable (in time).
2. Heap Management - A Selective Introduction

2.2.2. Space

When reasoning about the space dimension of a memory management system there are several key aspects one can observe.

**Total memory consumption.** The total amount of memory that is requested by a mutator program including the overhead needed by a memory management system.

While it may be interesting to know how much memory a program consumes, this property does not set different memory management systems into relation. The total amount of memory consumed by a mutator is the same, regardless of the underlying system’s allocation scheme.

**Live heap-size.** The amount of memory that is not available for further allocations from within the mutator at a time $t$.

In contrast to total memory consumption, the live heap-size shows the current state of the heap and thus includes not only the memory utilized by the mutator, but also any overhead that is related to the underlying allocator.

Despite the overall consumption of a mutator and the overhead of the underlying allocator, it may also be of interest how the requested memory is actually placed on the heap. The property to measure how well objects are organized on the heap is called fragmentation.

**Fragmentation [1, 7].** Heap fragmentation is the inability to acquire memory through an allocation request, because the already requested memory is arranged in a way that does not allow any further allocation. Fragmentation can be divided into internal and external fragmentation.

**External fragmentation [1, 7].** When contiguous memory is subject of allocating and freeing it gets divided into smaller pieces over time. These pieces can get small enough to be unable to satisfy any memory request.

Although the overall free memory in the system may be large enough, it is possible that a request cannot be served because there is no contiguous piece of memory available, that is large enough.
Figure 2.3.: External fragmentation

Figure 2.4.: Internal fragmentation

Figure 2.3 shows an example of external fragmentation. In this specific example the heap was filled with the following allocation sequence:

\[
a = \text{allocate}(1.5k) \rightarrow b = \text{allocate}(1k) \rightarrow c = \text{allocate}(2k) \rightarrow \\
d = \text{allocate}(0.75k) \rightarrow e = \text{allocate}(1k) \rightarrow f = \text{allocate}(0.75k)
\]

Afterwards the heap was partially cleaned by several deallocation calls:

\[
deallocate(b) \rightarrow deallocate(e)
\]

The heap is then left in a state where it is unable to serve a 1.5k allocation request, although the overall free memory is 2k, due to external fragmentation.

**Internal fragmentation [1, 7]**. A memory management system usually (not always) acquires not exactly the requested size of memory \(s\), but a block of a size \(b\) (due to padding, performance, … reasons). The amount of memory that is lost \(b - s\) is called internal fragmentation.

Figure 2.4 illustrates an example of a 1.5k allocation request on a 2k size class (block-size). The 0.5k internal fragmentation will be temporarily lost until the object is reclaimed by the allocator.

Fragmentation can be reduced by mechanisms like splitting and coalescing of blocks, or moving used memory together into an area. The procedure of moving used memory together is called compaction. When memory of the same size is allocated and freed in a cyclic scheme, it is possible that the heap is self-compacting, meaning that allocation
requests always end up in the same slots that have recently been freed by the allocator.

Like in the time dimension there exists the analogous term predictable (in space). For the space dimension this refers to a system that is capable of reporting its internal state regarding fragmentation and used/free memory at any point in time.

### 2.3. Explicit Memory Management

Explicit memory management systems are also referred to as plain allocators [1]. These systems provide an interface for acquiring and releasing memory in form of chunks or objects. The programmer is in charge of (almost) all tasks and thus these systems are error prone. However, they may be used to create well-performing systems, because the programmer has to control each part of it.

Listing 2.1: Allocator interface

```c
void init (size_t size);
void* malloc (size_t size);
void free (void* ptr);
```

Listing 2.1 shows the interface an allocator may provide. Although the naming of the functions in this listing is based on the libc interface of the programming language C (a library that provides “systems calls” for Unix-like operating systems), similar functions exist for every explicit memory management system.

The `init` function is called before any other allocator function and is in charge of bringing the memory management system into a consistent state, i.e. initialize data structures. It also acquires a chunk of memory from the underlying operating system and fits it into its data structures. Depending on the implementation, the call itself may be issued by the programmer or be implicitly called by the allocator. In principle, it can also be delayed upon the first allocation call. After successful initialization of the allocator, the programmer may utilize `malloc` and `free` functions to acquire and release memory, respectively. The function `malloc` takes the size of the requested object or memory as parameter and returns a pointer to the chunk of memory that has been acquired by the mutator. An object-oriented equivalent (e.g. `new` in Java) would take the object type as parameter, indicating the size implicitly. The `free` method is then used to return the memory from the mutator back to the allocator, i.e. perform a deallocation.

There exist several different types of explicit allocators that can all be distinguished by the underlying approach to manage the heap and the requests. The central problems are performing requests fast and managing the fragmentation on the heap. Ideally, a memory management system would react to requests in a way that it takes the memory where it would later on cause no fragmentation. Since this is not possible in a general case, because a program cannot anticipate how much memory will be consumed and how the allocation and deallocation theme will look like, allocators do follow different strategies that approximate certain behaviors (i.e. application domains) best. An introduction to strategies, such as best-fit, worst-fit or segregated free-lists, can be found in [1]. A more elaborate description and comparison of these algorithms and strategies can be found in [9].

2.4. Implicit Memory Management

Implicit heap management, also called garbage collection [4, 10, 11, 12], is a type of automated MM. Like in explicit heap management, the programmer utilizes an allocator to acquire memory from the management system. While in the explicit approach the programmer is also in charge of freeing unused memory, the implicit approach uses a garbage collector to accomplish the management of the heap. This may happen either by utilizing the deallocation function of an underlying allocator, or by reorganizing the heap in some other way. In this implicit case the deallocation function (free) is not visible to the programmer.

It is also possible to mix both approaches. For instance, one could define a memory region (in terms of scope) where a GC operates and leave all other objects not allocated in this scope to an explicit heap management system [13].

Independent of its type, every GC operates in two different steps [4]:

1. **Reachability analysis**

2. **Collection**

   Ideally, every GC would be able to determine the needed-set of objects that have been allocated. Since it is generally not possible to guess the semantics of variables, every GC calculates an over-approximation by performing a reachability analysis, as already
described in Section 2.1. The previously defined *live objects* can now be extended with *dead objects* that are the counterparts, i.e. objects (memory) that are not reachable. Bacon et al. [4] also describe them as matter (live objects) and anti-matter (dead objects). Depending on the type of GC it operates on either the matter or the anti-matter, or even both [4].

### 2.4.1. Why? - Benefits and Drawbacks over Explicit Memory Management

Before describing the different types of garbage collectors, it is already possible to summarize the basic benefits and drawbacks of such systems by assuming that the principle steps are always performed right.

Garbage collection can be used to eliminate certain kinds of bugs:

**Dangling pointers** [10]. In the explicit approach it is possible to free memory, even if there are references pointing to it (memory is therefore still classified as being needed). The dangling pointer then references a block of memory that could already be reused, and thus changed. This eventually results in program crashes or even worse, unexpected behavior. The phenomenon of dangling pointers is not possible when using a GC, since memory is only reclaimed if there exist no references to it.

**Double free** [10]. A double free is actually a subtype of a dangling pointer and occurs when a chunk of memory is freed multiple times and usually results in a program crash. As a garbage collection system keeps track of free and occupied memory, this does not happen.

There also exist bugs that can not inherently be removed from a system but be reduced:

**Memory leaks** [10]. A memory leak is free, unused memory that has not been reclaimed by the underlying heap management system, which can either be controlled by the programmer or a GC. Garbage collection can reduce memory leaks by eliminating certain types of it, i.e. a GC can detect memory that has no references pointing to it from the main program and is thus able to reclaim it. This type of leak is called *physical memory leak* [14]. In contrast to this, a *logical memory leak* [14] is a leak where the main program still contains a reference to an object/memory. This can happen if a programmer forgets about an acquired object and thus also forgets to override/unset the reference to it. Since
it is not possible to guess the semantics of any reference in the program in general, a GC is not able to avoid such a leak by reclaiming the memory.

There exists a further memory leak, called cyclic memory leak, which is only relevant for reference counting GCs. For more information on this topic, see Section 2.4.2.

Of course these benefits come with the drawback of an additional system interfering in the program, costing resources. A running GC not only costs CPU time, but may also need bookkeeping structures that are allocated on the heap themselves. Furthermore garbage collection introduces pause times that may be critical for real-time oriented systems.

2.4.2. Garbage Collector Types

Depending on the type of the GC the previously mentioned steps result in different models of operation. Literature mentions basically two different types of garbage collection systems, namely, tracing and reference counting GCs [4].

Note: Although both have benefits and drawbacks, tracing GCs will be used for further explanations after a short introduction on both.

Tracing Garbage Collectors

Tracing GCs perform an explicit reachability analysis of all objects by using a batch based approach. This means that they scan the heap for reachable objects in the first place. In order to do so, the GC starts with a known set of objects that have to be reachable, called roots. Depending on the usage domain (real-time, incremental or concurrent), the definition of which objects may belong to the roots may vary. One possible definition is [10]:

• **True roots**: CPU registers and global variables
• **Fast variables**: Stack frames and variables that have been allocated in there
• **Slow variables**: Anything else that is referenced by the others

Depending on how the reachability analysis is performed, true roots and fast variables can be combined into one group and are then called roots.

After specifying what the root set of objects is the tracing GC starts the reachability analysis. This step is also called marking phase. There exist several different styles of
marking, all having their advantages and disadvantages. What they have in common is that they follow pointers from the root set and mark objects that can be reached as live objects.

The collection step is then performed on the remaining objects that have not been marked (dead objects). Again, the actual implementation depends on which collection strategy is used.

**Reference-Counting Garbage Collectors**

Reference-counting follows a different approach. Instead of “cleaning the heap by shoves” the GC keeps a counter at each object indicating how much references are currently pointing to it. As a result these GCs are incremental by design. The general steps, reachability analysis and collection, are not performed on the whole heap, but on single objects.

In order to keep this reference up-to-date, even in a concurrent environment, a reference-counting system needs to rely on write-based memory-barriers [4]. This means that whenever a pointer variable is assigned the barrier enforces a serialization of reference counter processing.

**Listing 2.2: Cyclic memory leak algorithm in Go**

```go
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
1 type A struct {
2   b *B
3 }
4
5 type B struct {
6   a *A
7 }
8
9 func main() {
10   var a = new(A)
11   var b = new(B)
12   b.a = a
13   a.b = b
14   a = nil
15   b = nil
16   // leak
17   ...
18 }
```

Cyclic memory leak. As previously mentioned, reference counting garbage collectors may suffer from cyclic memory leaks. Listing 2.2 shows a possible leak that is illustrated using the programming language Go. The main program uses the memory allocator to create objects for the structs A and B. It then creates a reference cycle by pointing to each other. Finally, the main program overrides the “main” references with nil, leaving the objects behind with their reference counter still being > 0.

These types of leaks can be avoided if the GC uses so-called cycle detection.

Tracing vs. Reference-Counting

Although both approaches achieve the same result, i.e. a managed heap, they are different and thus have different usage domains. Table 2.1 shows the major differences [4]. While tracing garbage collectors tend to be throughput-oriented because of the batch-like processing, reference-counting systems are more suited for applications that need real-time capabilities. Although the cost per mutation (of a reference) is higher using reference-counting (barrier’d counter processing), the pause times tend to be shorter because objects are handled when they become unreachable. Thus a reference-counting GC is also deterministic regards the cleanup of unused objects. In contrast to this a tracing system is non-deterministic because there exists no guarantee until which time (or even if) an object is freed.

<table>
<thead>
<tr>
<th></th>
<th>Tracing</th>
<th>Reference counting</th>
</tr>
</thead>
<tbody>
<tr>
<td>Collection style</td>
<td>Batch</td>
<td>Incremental</td>
</tr>
<tr>
<td>Cost per mutation</td>
<td>Low</td>
<td>High</td>
</tr>
<tr>
<td>Throughput</td>
<td>High</td>
<td>Low</td>
</tr>
<tr>
<td>Pause times</td>
<td>Long</td>
<td>Short</td>
</tr>
<tr>
<td>Real-time</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Collects cycles</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>

Table 2.1.: Comparison of tracing and reference-counting GCs [4]

Go does not suffer from this leak, because it uses a Mark-Sweep GC, but more on that later in Section 2.4.4.
2.4.3. Tracing Garbage Collection Characteristics

In addition to the previously depicted fundamental difference in garbage collection there exists several variants of tracing collectors that need to be mentioned briefly in order to understand the implementation that follows the theoretical concepts. Note that while these variants can also be used for reference-counting GCs, they are described with respect to tracing collectors because the Go runtime currently implements a tracing GC.

Moving vs. Non-moving Garbage Collectors

After determining the live and dead objects, in the non-moving approach the GC deallocates all dead objects in the collection step. In the moving approach the GC copies all reachable objects into another area on the heap. While this seems to add a significant amount of overhead, because copying involves costly instructions, this approach nevertheless has some advantages:

- Dead objects do not have to get reclaimed because the entire area they reside in can be cleared.
- External fragmentation can be minimized because objects are copied altogether into a new area, organizing everything from scratch.
- Allocation of objects is fast because the allocator does not have to organize anything to reduce external fragmentation. A pointer that points to the next free element serves as allocator. Upon allocation this pointer is moved to the next free block. This concept is also called bump-pointer.

Fenichel and Yochelson [15] present how basic moving systems, i.e. tracing semi-space collectors, work in more detail. Further improvements are illustrated in [16].

Stop-the-world – Concurrent – Incremental

Another distinguishing category for tracing garbage collectors is their mode of operation. The stop-the-world approach [11, 4] implements a barrier that stops the mutator completely, hence the name. The obvious disadvantage here is that the memory manager cannot be used in any real-time systems that have the requirement of meeting hard deadlines. Even user-interaction may suffer from this bad latency behaviour. The benefit is that...
the collector has full control of the heap and can utilize the hole computing power to get its job done.

Other approaches are *incremental* and *concurrent* ones. Incremental GCs execute their phases in intervals, permitting mutator execution in-between these intervals. While this may be hard to get right, it preserves latency requirements of a mutator. The first example for an incremental marking algorithm is the tri-color marking described in [17]. When having a GC that is run in its own thread of execution, this is called concurrent garbage collection. Since this is not the topic of this thesis, it will not be dealt with in more detail, but further information on it can be found in [18].

**Conservative Reference Detection**

Another property of tracing GC is how they handle the detection of references that refer to other objects. In order to do a proper tree-traversal in the reachability analysis, it is important to know when a reference is encountered while investigating the attributes of an object. In a non object-oriented environment this also holds, apart from the fact that it is not iterated over the member attributes, but over fields in the memory (usually word size).

![Conservative Reference Detection](image)

Figure 2.5.: Precise and conservative reference detection

It is not important whether an object-oriented or non object-oriented language is used, but how the internal values are represented. If it is possible to determine whether a field is a pointer or a value, then it is also possible to traverse the tree correctly. Figure 2.5(a) illustrates the approach in a language such as Java. The member attributes of objects in
2. Heap Management - A Selective Introduction

Java are not only strongly typed, i.e. it is clear for every location whether it is a pointer or not, but they are also reordered. In this example the references are copied to the end of the object making it easy to iterate over them and proceed with the tree-traversal. As this approach is completely accurate, it is called precise GC.

In contrast to this, languages such as C do not have this strict organization in their memory layout. In fact, C does not even provide a strict type for claimed memory. An allocation request in C just gets the amount of memory that is needed, reserves the block and then returns the pointer. It is thus not possible to iterate over fields in the memory block, but instead a loop over blocks of memory is created. In this, each block is exactly one word in size. Figure 2.5(b) shows the memory layout of such a language. Each memory location (word) is then interpreted as if it were a pointer. If the location reaches into the upper and lower bounds of the heap, which are recorded by the allocator, it is assumed that this is a pointer. Furthermore, the mark bit for the corresponding location is then set by the GC. As this is overall an overapproximation of possible pointers, this algorithm is called conservative GC [19].

2.4.4. Naïve Mark-Sweep Garbage Collection – An Example

Finally, this last section provides an overview over a naïve mark-sweep garbage collection [4, 7] system. This is the same type of collector that is used by the runtime of the Go programming language, hence the example.

As mark-sweep collectors are tracing GCs, they use a reachability analysis to determine live objects. This step is called marking phase. In a second step these collectors then get rid of dead objects in a collection step. This is called sweeping, hence the name mark-sweep collector. For simplicity, the mark-sweep collector presented here uses a non-incremental, non-moving, stop-the-world approach, as already previously described. More sophisticated approaches are of course possible and can be found in [20].

Figure 2.6 illustrates the general tasks that take place as soon as a garbage collection run is triggered, for instance when the heap is “full” and the underlying allocator is unable to respond to an allocation request.

In the reachability analysis, the GC marks objects as live, traversing an object graph starting at the root set. The mark-state is usually recorded in a header field, or some other field responsible for meta data information that is somehow connected to the object. In
order to traverse the graph, the mark-sweep GC investigates the internal references of the object (chunk of memory) and, if possible, follows their pointers. The references can either be defined, like in Java, in an object header, or detected in a conservative approach, as previously described. Figure 2.6(a) shows the initial set of objects that have then been marked in Figure 2.6(b). The objects $E$, $H$, $I$ are not marked as they are not reachable from the root set within some sequence of references. The objects $A \rightarrow B \rightarrow D \rightarrow C$ and $G \rightarrow F$ are marked as live.

The second step is then illustrated in Figure 2.6(c), where the GC starts iterating over the heap and frees all objects that have not been marked. Furthermore, the collector also removes the marks of all live objects to ensure a consistent state after one collection run. The final state, depicted in Figure 2.6(d), shows the graph after the dead objects have been freed.

After successfully freeing all dead objects in the sweep step, the GC is finished and the mutator can continue to execute. Mark-sweep collectors suffer from the already mentioned drawbacks of high pause time that result in bad latency for the running mutators. On the
other side they provide high throughput as no further interactions, i.e. references counter updates, are needed.
3. Short-term Memory (STM)

short-term memory (STM) [2] is an implicit heap management model that can be seen as an alternative to the explicit malloc/free model and the standard implicit garbage collection based approach. Unlike in explicit malloc/free systems or implicit garbage collection based systems, the objects allocated in the STM model do not live forever, but have an expiration date instead (see below for the definition). This means that an allocated object is only guaranteed to exist (and be valid) for the amount of time that has been specified. The lifetime of an object, after it has been allocated with a specified expiration date, is then defined by the program time and its progress. It is possible to control this program time. When time advances fast, objects will expire faster and therefore be deallocated faster. This results in a lower overall memory utilization. In contrast to this, it is possible to let the time stand still. This represents a system where no deallocation takes place.

3.1. Expiration of Objects

Before going into more details on the model, it is important to explain how objects expire in the STM model.

Expiration date. An expiration date is the date (timestamp) until when the object that it has been assigned to is valid, i.e. guaranteed to exist in memory.

As expiration dates of objects can be freely chosen, one may use different approaches to specify them, depending on how much knowledge on the program is available.

With full knowledge of the program, one can allocate an object with the exact expiration date. This is also safe because an object will only be deallocated after it expires. This basically represents a malloc/free based model, although it is different in its usage. While it may be more difficult to control the progress of time to get the exact behavior regarding the expiration date, the short-term based approach requires no pointer to deallocate the
3. Short-term Memory (STM)

object since everything is performed implicitly. Figure 3.1 shows an allocation sequence using exact expiration dates for two objects, allocation(3) and allocation(7). Note that while the objects are meant to survive 3 and 7 time units respectively, the first time unit has already started for both objects, since the current time has not (yet) advanced.

![Figure 3.1. Allocation with known expiration date][2]

In contrast to specifying exact expiration dates, objects can also be allocated for exactly one time unit. This approach may be useful when there is little to no information available on the program behavior. When time progresses all allocated objects expire at once. Figure 3.2 shows such an allocation of an object including its life time. While this strategy is safe and may be used without much knowledge of the behavior, it suffers from inefficiency because the object resides in the memory until the time advances although it may already be unused for some time.

![Figure 3.2. Allocation for one time unit][2]

Depending on the knowledge of the program, the programmer might choose either one of these extremes, or estimate the refreshing of the objects or the ticking of the clock in some way. Because it may not always be possible to estimate the time that an objects needs to be live, it is also possible to extend the previously set expiration date of an object using another refresh call. This extended refreshing mechanism can either be utilized by a programmer, or a runtime system using STM as its collection strategy for objects.

Figure 3.3 shows an example of an object with an overall lifetime of seven. While the object has only been allocated with an expiration date of 2, the two consecutive calls refresh(3)
3. Short-term Memory (STM)

and refresh(2) have extended its lifetime to 7. In this example, it is important that the extending refresh statements have been executed in the last interval before they would have been expired, as the expiration dates of two consecutive refresh calls don’t sum up, but replace the currently existing ones.

![Figure 3.3.: Allocation of an object for seven time units using further refreshing][2]

### 3.1.1. Liveness - STM vs. Traditional Malloc/Free

As previously described, refreshing can be used to extend the lifetime of an object for a specified amount of time. The focus of STM is therefore on objects that are currently in the needed set of the application, with possibilities to extend their lifetime. In the traditional malloc/free model the approach is different. Instead of concentrating on the needed set, the programmer is forced to think about which objects are not needed anymore. A free call basically sets the expiration date of an object to the current time. Because of the fact that the programmer has to be interested in not-needed objects, it may also be the case that the object which is to be freed cannot be accessed anymore. This, however, cannot happen when using STM, as the only objects that are currently valid, i.e. have an expiration date later than the current time, can be extended. These kind of objects will always be reachable.

Figure 3.4 illustrates the different kinds of object sets that may reside on the heap. While from the programmer’s perspective only the needed set is of interest, he may be forced to think about objects that are not needed but still reachable, in order to correctly free them. Figure 3.5 shows the sets over the program time. The time frame between allocation and last use is the one that the programmer is interested in, i.e. the time used for calculations. The blue time frame indicates the time when an object is still reachable, but not needed anymore. Basically, the collection system used (being the programmer for malloc/free) is
3. Short-term Memory (STM)

Figure 3.4.: Different sets of objects on the heap [3]

in charge of freeing the object in this time interval.

3.2. Model

For using STM, i.e. programming with it, a fully backwards-compatible model has been developed. In order to provide this backwards-compatibility, every object that is allocated is managed by the underlying heap management system per default. This may either be a GC for implicitly managed languages, such as Java, or the explicit malloc/free systems in C.

At any time after allocation and before deallocation, the object may then be transferred to the STM environment using a refresh(o, e) call, where o is the object and e is the expiration extension with $e \geq 0$. In order to preserve scalability it is important that this refresh-call is constant in time and lock-free (see Section 4.5 for an analysis of the implementation).

The result of this refreshing is that the object is flagged as short-term with an expiration date $(l + e)$ where $l$ is a software clock represented by an integer counter. The software counter is thread-local, meaning that every thread of execution maintains its own clock that is used as a basis for the object expiration. After the object has been flagged, it is managed by STM and not by the default system anymore. The object o is then guaranteed to exist
3. Short-term Memory (STM)

until the containing thread advances its (thread-local) time to \((t + e + 1)\) by incrementing the value of the counter through \((e + 1)\) \texttt{tick}()-calls. Again, these \texttt{tick}-operations need to be constant-time and lock-free to preserve scalability.

Note that an object that has been flagged as being short-term once, cannot be transferred back to the default memory management system. Although the design of such a call would not be too difficult, there have not been any use cases that needed this kind of operation.

### 3.2.1. Single- and Multi-Refreshing

For one thread of execution the previously explained rules for refreshing, ticking and expiration of objects apply. The situation is different when multiple threads of execution are involved. For the sake of simplicity, all threads of execution will from now on be called threads or goroutines (i.e. the Go equivalent – differences will be explained).

In addition to the already explained behavior, an object may also be refreshed by multiple threads, even in between ticking of clocks. This results in multiple expiration dates being created for a single object. Finally, it is possible to define the underlying invariant and rule of STM.

**Invariant [2].** An object in STM expires if all its expiration dates have expired.

**Rule [2].** An expiration date has expired if its value is less than the (thread-local) time of the thread that created the date using a refresh-call.
3. Short-term Memory (STM)

This invariant and rule do not interfere with the already stated descriptions in the introduction, as multi-refreshing from within the same thread has no effect on the liveness of the object, however only wastes CPU cycles. This approach is different from the explicit malloc/free model in the sense that in malloc/free systems it is an error when an object is deallocated multiple times. Likewise, it is also an error in STM when expired objects are refreshed.

In contrast to the first use case, multi-refreshes distributed among multiple threads indicate that an object is shared between these threads and needs to stay alive until it is expired in all threads. This implies that STM managed threads need some synchronization in order to create a global state that shows the expiration status. The following section on global time and Go channels illustrates how this coordination could be done.

3.2.2. Global Time compared to Go Channels

In order to handle globally shared objects, either some global time management or other synchronization methods are needed. First, a simple model of global time is illustrated. While this model only handles a fixed amount of threads and cannot deal with faulty (non-advancing) threads, it serves the purpose of explaining how this problem could be solved in principle. The second part shows one possible solution avoiding the problem completely using special programming techniques provided by the programming language Go.

Global Time

It is assumed that the number of threads is fixed by a number $t$. There may not be any dynamically created or terminated threads that are using STM managed memory. In order to do the synchronization, further Application Programmable Interface (API) calls are introduced. Namely, these are global-tick() and global-refresh(). The thread-local variables also need to be extended by an integer counter holding a thread-local global time $t$ (phase). Additionally, a global variable needs to hold an integer count that represents a synchronized global time $g$. A variable holding the fixed number of threads is also needed. The local refresh and tick calls still work as they have been described. The additional global methods, however create a so-called global scope. The global-refresh call creates a global expiration date that behaves the same as the normal expiration date, except that the object expires regarding the global clock. The global clock is represented
3. Short-term Memory (STM)

Figure 3.6.: Global-time ticking [3]

by the global time $g$. The synchronization happens within the global time phase that is
stored thread-local. The intention is that each thread has to have the chance to refresh
the object regarding the global clock in one time cycle. The initial state is that $t = g$, i.e.
the global phase of the thread corresponds to the global time. Whenever a global-tick
call is executed, the global time $t$ of the thread is incremented, if (and only if) $t = g$. If
those two are already different, then the function returns immediately. Furthermore, the
globally-ticked threads counter is decremented in an atomic way. When the ticked threads
counter reaches 0, i.e. when all threads have performed a global phase, the global time is
incremented and the thread-local global phase counters are set to the global time ($t_i := g$).
Figure 3.6 illustrates this behavior.

Go Channels

Although the capabilities of Go haven’t been elaborated so far, this section provides an
outlook and introduces the term channel. A channel in Go is a medium through which
multiple threads, so-called goroutines, can exchange information to each other. All locking
and synchronization is provided by the Go compiler and runtime environment.

It is therefore possible to simplify the solution for multiple threads of sharing an object.
Instead of keeping the object global, it is possible to send it to another thread. This creates
another “copy” of it (see following sections that describe the Go internals) that can then
be used by the receiving thread. The invariant that needs to hold is defined by:

**Invariant.** An object that is received through a Go channel is subject of the default memory
management system.

As a result, receiving objects have to be transferred to STM again. Of course channels
3. Short-term Memory (STM)

could also be used to transfer the expiration information of the sender, i.e. local clock state and expiration extension, which could then be used by the receiver. However, this remains an option for the receiver and is not enforced.
4. Short-term Memory Implementation in Go

This chapter deals with an implementation of the previously described STM concept using SCM in the programming language Go. Therefore the first part provides an introduction to the Go language in general. This includes language specific aspects, as well as, an implementation overview of the runtime library. The introduction that is provided on Go’s capabilities, however, is not meant to be a complete overview, but should give the reader an idea of how the language is assembled regarding heap-based MM. Finally, the chapter introduces the concept of SCM as a possible approach for a STM capable environment.

Note that the full implementation can be found at http://tiptoe.cs.uni-salzburg.at/short-term-memory/.

4.1. The Go Programming Language - A Technical Examination

Go has officially been announced as a programming language in November 2009. It aims to be a general purpose programming language that can also be used for systems programming [21]. As a result, Go competes against languages such as C/C++. The language tries to combine the efficiency and safety of statically typed compiled languages with the ease of use of dynamically typed languages, such as Python. The key aspects of Go are [21]:

- Strongly typed
- Garbage collected
- Explicit support for concurrent programming
In order to provide these features, the Go compilers link all code against a runtime library. More on building Go programs can be found in Section 4.1.3.

4.1.1. Go Specific Concepts

Go aims to be especially useful for concurrent programming by supporting special primitives that offer synchronization and data exchange, while preserving a known state in all threads of execution. Go offers a new way for creating these threads of execution by introducing so-called goroutines. Goroutines can be compared to C threads [1] as they allow execution of different (or the same) functions in a concurrent way. Code can even be run in parallel, if there are enough CPUs to schedule them on. A fair user-space scheduler guarantees that all goroutines get the same amount of time for their execution. They are light-weight as they all share the same address space and only need a small stack to start from, as the stack is increased dynamically at runtime. The syntax to create them is defined by:

Listing 4.1: Goroutine definition [5]

\[
\text{GoStmt} = \text{`go`} \ \text{Expression} .
\]

Expression has to be a call. When invoking a function to create a goroutine the call is non-blocking, which means that the call immediately returns and the function is executed in the background [5]. The program’s entry point, the main function, is also a go routine that is started up by the runtime library.

While this construct solves the problem of creating concurrent threads of execution, it cannot be used for synchronizing and communicating data. The primitive that is used to achieve this synchronization is called channel. A channel is an object that can be used by multiple goroutines to send and receive information in form of other objects. Since calls are blocking per default (but can be used in a non-blocking, buffered way) channels can be used to wait for other execution threads. In fact, the Go authors try to propose the following idiom:

“Do not communicate by sharing memory; instead, share memory by communicating.” [21]

Ideally this would mean that all data that needs to be shared is sent over channels. While the Go authors state that this is the preferred way, it is not enforced by the language.
4. Short-term Memory Implementation in Go

It is still possible to use shared memory passing pointers to objects as parameters to goroutines. The send and receive operations are defined by:

Listing 4.2: Go sending and receiving [5]

1. SendStmt = Channel "'<-'" Expression .
2. RecvStmt = [ Expression (' '=' | ' ::= ' ) ] RecvExpr .
3. RecvExpr = ... "'<-'" PrimaryExpr ... .

The important thing here is that the keyword "<-" can be used to send and receive information based on if there exists an equality sign. The following chapter will deal with the creation of channels.

4.1.2. Heap Management in Go

Heap management in Go is performed using an implicit dynamic approach. This means that there exist calls to acquire memory, in the form of an object, for an arbitrary type. There is no way to fetch an unbound chunk of memory such as with malloc in C. In order to create an object in Go each allocation call needs a known type as parameter. Memory objects are removed by a collector when they are no longer needed. Go uses a mark-and-sweep GC as described in Section 2.4.4.

Go offers two slightly different allocation operations that may be used in programs to allocate objects [5].

Allocation using “new”

The built-in allocation function new(T) is used to obtain an object of type T and returns a typed pointer *T to this newly allocated object. I.e., this function can be used when obtaining new structs. This call can be compared to the new call in Java or C++. It is different to the C call malloc in the sense that it takes a type and not the number of bytes to allocate. Additionally, it returns a strongly typed pointer instead of the arbitrary typed void* in C [22].

Allocation using “make”

Unlike other systems programming languages such as C and C++, there exists a further call that may also be used to obtain objects. It is different from new in the way that it is
4. Short-term Memory Implementation in Go

used to obtain built-in object types. For instance, `make` can be used to acquire channels. Internally, this function creates and initializes the structures that are used by the built-in type (that wraps them). In contrast to this, `new` is just returning a pointer to the zeroed object (bound chunk of memory). The function `make(T, ...)` where `T` is a type construct, can be used when special built-in types are needed. Currently, there exist three built-in types [5] that can be used with this function.

The `map` data type is a hashmap to associate different types of keys to values. For instance, this type may be used to construct a dictionary with strings as keys and another type as value. The statement that generates this dictionary would be `make(map[string]type)`. The second internal type is `slice`. It wraps an array of a specific type with a structure that allows bounds checking. For a description of its internal layout see Section 4.2.3. The third internal type is the previously described `channel`.

4.1.3. Building and Deployment

Since Go aims to be a systems programming language, the code is compiled directly into a binary that is then executed on the specific platform and architecture. At the time of writing these platforms include 32 and 64 bit variants of Linux and Darwin. An unofficial Windows port is also available. The supported architectures are ARM, 386 and amd64 [21]. The compilers used to create a binary are available in two flavours. Either a derivation of the compiler from the Plan-9 1 compiler suite called 6g, or a GNU Compiler Collection (GCC) 2 fronted. While the GCC compiler creates intermediate code that may be linked against other programs, the Plan-9 derived suite relies on its own object format and a static linking approach (as per 5-March-2011). Compiled programs or libraries are then bound to a single hierarchy namespace, called `package`. In both variants a single internal package, called runtime, is always included in a program. This package contains a user-space scheduler to manage goroutines, an allocator and GC, the concurrent primitives and all built-in functions and types. The experiments used in this thesis rely on the Plan-9 variant of the compiler, since experiments showed that it is a lot faster when compiling its own runtime library and dependencies 3 4.

---

2 [GCC, the GNU Compiler Collection - http://gcc.gnu.org](http://gcc.gnu.org)
4. Short-term Memory Implementation in Go

4.2. Go Runtime

The following section deals with the internal runtime and how it is assembled including the following modules:

- A user-space scheduler
- An allocator
- A mark-sweep GC
- A provider for built-in functions
- A broker for concurrent data exchange and synchronization

Since only heap-based memory management is covered in this thesis, the focus of this section is on the allocator and GC. The runtime that has been used for the following examination has been tagged by the Go authors with the version `release.2010.11.02` and is available at the Go project site. At the time of writing there exists no substantial documentation on the runtime. The descriptions found here are thus interpreted from source-code.

4.2.1. Allocator

In order to provide the best performance possible, while still maintaining the concurrent usage possibilities, the allocator has been split into various levels of operation. As a result, calls to `new` or `make` do not always acquire memory from the operating system, but may use space available in a cache. Before giving an overview of the various caching levels, it is important to know how objects in Go are represented internally. Although in the user-land Go program there exists no raw memory, internally the objects are handled as chunks of memory, which are contiguously aligned blocks of memory. Before acquiring this memory, the different kinds of requests (i.e., make and new) are internally mapped to a single function called `mallocgc`. This function obtains memory of a given arbitrary size, and invokes the GC, if needed. Although the size of the object may be of arbitrary size, it is then mapped into size classes. There exist dynamically (at program start) calculated size classes for objects of a size below 32KB. Objects of a bigger size than 32KB are allocated

---

5 The Go Programming Language - http://code.google.com/p/go/
directly on the program’s heap without any cache interference.

As already mentioned, the object’s internal representation is just a pointer to the chunk
of memory it represents. Unlike other garbage-collected languages, such as Java, there
exists no internal object header that is prefixed to the actual object body. Go keeps track
of object-related information with a single status flag that is situated in a meta object.

**Memory Layout**

As illustrated in Figure 4.1, in Go each goroutine maintains its own cache for objects
throughout its lifetime. This cache is internally represented as array of MCache objects.
Each MCache contains a linked list with object-slots of the same size class, hence the
array. Whenever there is no object-slot available in the per goroutine cache, the corre-
sponding size class gets refilled with objects from the next level. This next level is called
MCentral. Again, this cache level is clustered into different size classes. Because this
level is program-global, meaning that every goroutine needs to access it in a concurrent
environment, there exists a lock to regulate these access requests. As a result, the MCache
routines acquire more objects than the needed amount, to amortize the cost for getting
a program global lock. As this is also basically just a locked array of linked object slots,
it can get empty. In the case that MCentral needs to be refilled, it acquires no objects
from a structure called MHeap. This structure finally uses syscalls to get memory from the
operating system.

As previously mentioned, Go keeps track of the status of an object by a single flag. This
flag indicates GC relevant information, as well as a finalizer state. It is not located in front
(or at the end) of the object itself, but it is situated in a meta object, called MSpan. This
MSpan is basically created by the MHeap structure, which also clusters the memory of a so
called MSpan into objects of a specific size class. It usually forms the meta object for a set
of pages, also called run of pages, of a given size class.

The object relations are illustrated in Figure 4.2. In order to find the corresponding meta
object for a given object, Go uses a three-level radix tree (also known as PATRICIA) [23]
to map pages to MSpans. The page of an object can be found by dividing its address by
the pagesize (which is a constant). After mapping the page to a specific meta object, one
can find out the index of the flag in the array by computing the index of the object in the
4. Short-term Memory Implementation in Go

![Go Memory Layout and Contention Levels](image)

Figure 4.1.: Go memory layout and contention levels

The base address is the starting page address of the run. The size of a class is stored in an array that is generated at the start of the program.

### 4.2.2. Garbage Collector

Go utilizes a mark-sweep GC, as described in Section 2.4.4. It is implemented using a stop-the-world approach, meaning that whenever the GC kicks in, all other goroutines are stopped. After the GC is finished with a full mark and sweep phase, the user program continues. Go does not use a timer or a separate goroutine to trigger its GC, but relies on the amount of already allocated memory to start a GC run. Whenever memory is acquired, resulting in a `mallocgc` call, the current state of the heap is analyzed and, depending on the size of the requested object and the memory state, a run is triggered.

A minor difference to a classic mark-sweep GC may be that Go’s GC operates on regions, represented as `MSpan`, to iterate over the objects. As Go keeps track of the allocated...
memory through its MSpans, it is enough to search the pages having such a meta object. Go uses a pessimistic (conservative) approach to search for references, by comparing each word to the minimum and maximum heap bounds. Whenever a possibly matching address is found, the block is marked as live. This, of course, may include false positives because structs may contain 64 bit unsigned integers that seem like addresses. The sweep phase then iterates over the MSpans again and frees every block that is not marked as live. The free may be delayed into the next GC run if a finalizer needs to be called.

### 4.2.3. Slices

As slices are commonly used data structures in Go, they are also integrated into STM. While this enables a much better programming experience, the library integration has to take care of the built-in type. Figure 4.3 shows the memory layout of a slice. Although a slice in Go is capable of holding an arbitrary type, the runtime uses the calculated raw memory size for its memory representation. A slice consists of 3 fields: a byte pointer (ARRAY) to the actual memory chunk, a current length (LEN) and a maximum capacity (CAP). The length and capacity are visible to a user program via the built-in functions len() and cap(), respectively. When dealing with STM, it is important to note that a slice that is created using make([]someType, capacity) allocates the internal struct containing the three maintenance fields and an actual buffer.
4. Short-term Memory Implementation in Go

The example in Figure 4.3 shows a slice holding 12 elements with a capacity of 16 of some arbitrary type.

![Internal memory layout of a slice](image)

4.3. Hybrid Self-collecting Mutators

SCM establishes a MM system that implements the STM approach described in Section 3. While a possible solution would be to offer built-in function calls that are then translated by the compiler into the necessary code, a simpler library-based approach has been taken. The advantage is, of course that the compiler can remain a separate program and all code additions can be made just to a library. As a result, this means that there cannot be any specific compiler optimization, which makes the approach slower than the compiler-based one.

This section further covers the data structures and integration approaches that have been taken to create a STM capable environment.

4.3.1. Application Programmable Interface

SCM is fully backwards-compatible to the already existing MM in Go (see Section 4.4.2). In order to achieve that, it is necessary to provide an API for transferring an already allocated block from the garbage-collected environment to the mutator-managed one. Listing 4.3 shows the currently used API.
4. Short-term Memory Implementation in Go

(a) **StmRefresh**. The function is used to transfer an object into the SCM-managed environment. The extension given represents the expiration extension $\epsilon$ which forms the expiration date $(l + \epsilon)$, where $l$ is the current goroutine local time.

(b) **StmRefresh0**. Provides the same functionality as StmRefresh with the only difference being that the expiration extension is set to 0 by default. This results in an object expiring when the next tick action is performed. While internally calling StmRefresh this function can be convenient when dealing with short-lived objects.

(c) **StmTick**. Advances the goroutine local clock by 1.

<table>
<thead>
<tr>
<th>Listing 4.3: Go SCM API [2]</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a) func StmRefresh(object interface{}, extension uint32)</td>
</tr>
<tr>
<td>(b) func StmRefresh0(object interface{})</td>
</tr>
<tr>
<td>(c) func StmTick()</td>
</tr>
</tbody>
</table>

It is important to note that while STM and SCM would allow a mode using the global time, the Go implementation uses only a goroutine local scope. This decision was made to simplify the implementation and keep all provided methods basically lock-free. The only lock that is needed is a program-global one for acquiring the goroutine-local DescriptorRoot. See the following section for further information.

4.3.2. Self-collecting Mutators – Data Structures

In order to provide an SCM behavior the data structures and constructs illustrated in Figure 4.4 have been implemented.

**Descriptor** [2]. A Descriptor object is a pointer to an object that has been transferred to STM. As all required information is located at the object (in its object header), the size of a descriptor equals the size of a pointer in the architecture, which may be 32 or 64 bit. Furthermore, it is only contained in DescriptorPages to get a connection from SCM to the user-land objects.

**DescriptorPage** [2]. A DescriptorPage is an optimization for the concept of descriptors. The page contains storage for a specific amount of Descriptor objects, making it possible to keep the Descriptor free of a link pointer. The size of a DescriptorPage
4. Short-term Memory Implementation in Go

Figure 4.4.: SCM data structures

(DescriptorPageSize) is fixed at compile-time. This optimization reduces the number of allocations that are needed when the transferring/refreshing of multiple objects takes place. When the pagesize increases, the allocations get less but the memory consumption may be higher, as the last page may not be fully utilized. DescriptorRoot [2]. A DescriptorRoot is a goroutine-local object that is used to manage goroutine-local objects. In order to manage these objects the DescriptorRoot contains various lists and buffers of DescriptorPages. There exists a round-robin buffer for objects that have an expiration date, that is the locally clocked buffer. The size of this buffer is fixed at compile-time. Whenever an object is transferred to STM, a Descriptor for it is contained in the corresponding DescriptorPage. A single-linked list is implemented for expired objects, which again DescriptorPages. A further array contains a pool of free DescriptorPages that can be reused when a new object is recognized as STM and no non-full page is available. Additionally, the DescriptorRoot contains another flag that indicates whether a GC run is generally needed. See Section 4.4.2 for more information on the hybrid operation.

Object header [2]. Unlike the Go memory layout of objects, the SCM approach uses an object header that is located in front of an object. This is necessary as it is impor-
4. Short-term Memory Implementation in Go

It is important to access various parts of information without doing any costly lookup operations. As already mentioned in Section 4.2.1, Go distinguishes between small objects (≤32KB) and large objects (>32KB). Since their relations to the allocator are completely different, meaning that the Go runtime handles them completely different (cached vs. directly on the heap), two kinds of object headers are needed. The restriction on the object header that has been set is that it may not exceed 64 bits, or one word per object, for small objects. Both headers include a counter of how many descriptors are pointing to the objects and a size class indicator $i$. The counter is needed for indicating that a block of memory is short-term, and it may be used in cases where an object can be made STM in different goroutines. The size class is mainly used to distinguish between small and large objects. A size class of 0 represents a large object, the $i^{th}$ size class otherwise. The memory layout is illustrated in Figure 4.5. The differences between small (a) and large (b) objects can be summarized as follows:

![Object header for "small" and "large" objects](image)

(a) In order to perform an internal `free` call on a small object without doing any further memory lookups, it is necessary to keep the relative offset (16 bits) to the GC flag maintained by Go. Overall, the header for a small object takes up 32 bits for the counter, 16 bits for the size class and 16 bits for the offset, making up 64 bits overhead.
per object (at the object). Although the size class would fit in 8 bits, it is expanded to 16 bits to preserve 64 bit alignment.

(b) As large objects take up a whole MSpan object, it is necessary to keep the pointer to this meta object. On a 64 bit platform this results in an overhead of 104 bits per object. As the object itself has to be bigger than 32 KB to fall in this category, it is negligible.

As shown in Figure 4.5, the process to determine whether the given memory handle represents a small or large object utilizes the size class flag. Go defines that an object with size class > 0 represents a small one, otherwise it is a large object. Since the size of the field where this size is stored is known (16 bits), the value to look after is positioned at object_address − sizeof(sizeclass).

For information on the memory layout modifications and speed considerations see Section 4.4.1.

4.4. Runtime Modifications and Compatibility

This section describes the actual changes that have been made to the original Go runtime in order to create an environment that is capable of managing STM objects. Basically, this means that the memory layout needs to be changed in a way that it is possible to store the SCM relevant information. The second part deals with the compatibility to existing programs and how a hybrid operation can be achieved.

4.4.1. Memory Layout

As already mentioned, the memory layout has to be changed to create storage for SCM relevant information. Since each goroutine has to be aware of the locally managed objects, a single DescriptorRoot instance needs to be attached to each thread of execution. This is achieved by including the root in the internal goroutine management struct. The runtime internal scheduler therefore acquires a DescriptorRoot from a global pool whenever the goroutine is created. If the pool is empty, a new root struct is created. This approach guarantees that whenever a goroutine is run, SCM bookkeeping storage will be available. Contrary, the DescriptorRoot is returned to the global pool on destruction/termination of the goroutine. While in principle this global pool would have to be guarded by a program
global lock, only one goroutine at a time will ever be scheduled by the runtime. As the acquisition of the DescriptorRoot takes place in the scheduler itself, the pool is already locked by the runtime.

A further modification to the runtime is needed in the algorithms of the provided allocator that is internally used to fetch chunks of memory for user-land objects. As already stated, SCM needs an object header in front of the object to perform certain internal operations, such as free, as fast as the built-in GC.

Go’s GC performs its sweep freeing objects not entirely on an object level, but uses the meta-structure MSpan to loop over all objects contained by this run of pages. As the GC reference flag is also stored in this MSpan, it has fast access to reference and finalizer information. While looping over the object slots in the span, the GC then checks the corresponding flag if the object needs to be freed, or if a finalizer needs to be run.

In contrast to this approach, SCM operates completely on object level. This means that it has no direct access to the GC flag. Instead of looking up the MSpan meta object of the object through the radix tree, the object header is used to store a relative offset to the flag. The size of the offset field can be calculated by (4.2):

\[
gc_{offset}\_size[\text{bit}] = \text{ceil}(\log_2(#p * ps))
\]  

(4.2)

In the default runtime configuration, a run of pages MSpan can span at most 11 pages, limiting \( #p \) to 11. The size of a page \( ps \) is fixed with 4096 bits, resulting in a field size of:

\[
gc_{offset}\_size[\text{bit}] = \text{ceil}(\log_2(4 * 4096)) = 16
\]  

(4.3)

The header also contains the size class, since Go’s internal free function needs this information. The header itself is then placed before each object, by modifying the runtime allocator as illustrated in Figure 4.6. After the memory (some pages) is internally allocated from the heap, it is linked to its metastructure MSpan and assigned a specific size class. The memory is then split into equally sized blocks of the size that is indicated by its size class. To create place for an object header this linking procedure has been altered to keep a hole in front of every memory block. This slot creates storage for precisely one object header. The address of the header can then be calculated by:

\[
\text{header} = \text{object}\_\text{address} - \text{sizeof(object}\_\text{header})
\]  

(4.4)
This approach makes all necessary information completely available as soon as a reference to the object is available, which is always the case, since it is needed for a free anyway.

### 4.4.2. Hybrid Operation

In order to provide a backwards-compatible approach it is necessary to transfer objects from being managed by the default system to STM. As already illustrated in Figure 4.2, Go keeps track of the GC information in a single flag called $gcref$. The non-modified GC inspects these flags and compares them against so-called Ref-flags. If $gcref > 0$ then this object is currently managed by the GC and may or may not have references to it. A flag of 0 indicates a free memory block. It is thus enough to set this flag of an object to zero in the first refresh operation.

This approach has the drawback that it is not possible to transfer an object back to being managed by the GC. In order to do so, a storage for these flags would be needed. However, there has not been any use-case that needed this behavior.

Another important factor in the hybrid operation of the GC with STM is the fact that GC runs
may still occur due to the trigger that is used in the Go environment. In Go, every \texttt{mallocgc} triggers a check whether a GC run is needed. Since objects are initially managed by the GC, and only transferred after allocating, it is important to include a mechanism that prevents this check from triggering a collection run. As one of the goals of STM is to be used for short-living objects, a flag to recognize a pattern of \texttt{mallocgc} $\rightarrow$ \texttt{refresh} $\rightarrow$ \texttt{mallocgc} $\rightarrow$ \texttt{refresh} $\rightarrow$ $\ldots$ has been implemented. The flag, a boolean indicator, resides in the thread-local struct of a Go routine. Any refresh sets it to false and thus prevents the next GC check in \texttt{mallocgc}. The flag is set to true after the check has happened in \texttt{mallocgc}.

This workaround could be prevented by including allocation calls for STM in the compiler.

### 4.5. Implementation and Complexity

After the general modifications regarding memory layout in the runtime have been explained, this chapter now deals with the actual implementation of STM using SCM and the complexity of the corresponding algorithms. The goal of the SCM implementation as provided below is a constant complexity $O(1)$ for the \texttt{StmTick}, \texttt{StmRefresh} and \texttt{StmRefresh0} operations. Furthermore, all modifications in the runtime do also have to be constant-time to provide a scalable system. The resulting implementation aims at having the same complexity as the underlying allocator.

Note that, as already mentioned, the Go runtime is programmed in several different languages, i.e. C, assembly and Go. As a result, most code is written in C, with a few interfaces and algorithms expressed in assembly and Go for simplicity.

Furthermore, not all functions are explained in full detail as this would be out of the scope of the thesis. However, all necessary details to understand that the implementation complies to the model and results in the state complexity is described.

#### 4.5.1. Allocation Modifications – \texttt{malloc}

While in principle the STM approach is independent from the underlying allocator, it still needs to be embedded into it to initialize all previously mentioned header fields. Every
allocation request made, no matter if it is a new or make, ultimately calls the mallocgc function with the appropriate size (not size class) to allocate a block of memory.

Listing 4.4: mallocgc modifications

```c
void *mallocgc(uintptr size, uint32 refflag, int32 dogc, int32 zeroed) {
    ...
    if (size <= MaxSmallSize) {
        ...
        v = MCache_Alloc(sizeclass);
        OBJECT_HEADER(v)->sizeclass = s->sizeclass; // >0 = small object
        OBJECT_HEADER(v)->gcref_offset = (uint16)((uintptr)ref-(uintptr)v);
        OBJECT_HEADER(v)->descriptor_counter = 0;
    } else {
        ...
        size += ScmObjectOverhead;
        ...
        s = MHeap_Alloc(npages);
        v = (void*)(s->start << PageShift);
        v = (byte*)v + ScmObjectOverhead;
        OBJECT_HEADER_LARGE(v)->sizeclass = 0; // 0 = large object
        OBJECT_HEADER_LARGE(v)->span = s; // span is needed for free
        OBJECT_HEADER(v)->descriptor_counter = 0;
    }
    ...
    return v;
}
```

Listing 4.4 shows the modifications made to the mallocgc function. MaxSmallSize is the defined maximum size for which objects are sorted into a size class. Basically, the allocator differs between two algorithms:

1. \( \text{size} \leq \text{MaxSmallSize} \). Here the modifications for SCM set the appropriate header fields, as illustrated in Figure 4.5. In detail, it sets the size class to the Go internal one, the descriptor counter to 0 and the garbage collector flag offset, as already described.

2. \( \text{size} > \text{MaxSmallSize} \). The memory is directly allocated from the heap. Therefore, at first the size is increased by the large object-header. Secondly, after the allocation, the starting pointer is moved by the object-header size and again the properties of
the header are set. This time, the span (meta information) pointer is stored in the header, to be able to free the block later on.

Complexity

The complexity of the SCM implementation is basically dependent onto the general allocator implementation of Go and its modifications (4.5).

\[ O(mallocgc_{new}) = O(mallocgc_{old}) + O(malloc\_modifications) \]  (4.5)

Furthermore the complexity of the modifications shown in Listing (4.4) can easily be determined since they only contain dereferences of pointers and assignments with either pointer additions or fixed values, which are all constant. The complexity of the different size classes is shown in Formula (4.6).

\[ O(malloc\_modifications) = \begin{cases} 
O(1) & \text{size } \leq \text{MaxSmallSize} \\
O(1) & \text{size } > \text{MaxSmallSize}
\end{cases} \]  (4.6)

Since the complexity is constant, the overall complexity of \( mallocgc_{new} \) is also not changed, as shown in (4.7). The runtime behavior suffers from a constant overhead.

\[ O(mallocgc_{new}) = O(mallocgc_{old}) + O(1) = O(mallocgc_{old}) \]  (4.7)

4.5.2. Allocation Modifications – mcentral

Whenever the allocator requests objects of a given size class, the cache is first inspected for free objects. If the cache is empty, the central free list storage is asked for objects to refill the cache. If this queue is also empty, the central repository of objects is refilled. The runtime therefore acquires memory from the heap and slices it into several smaller blocks. The SCM system interferes here by increasing the slice size to store the (small) object header. The function responsible for these tasks is \texttt{MCentral\_Grow}.

Listing 4.5: \texttt{MCentral\_Grow} modifications

```
static bool MCentral_Grow(MCentral *c) {
    ...
    // Carve span into sequence of blocks.
```
Again the modifications are shown in Listing 4.5. The main modifications here are that the garbage collector array start is set accordingly, meaning that its offset changes from \(size \times n\) to \((size + ScmObjectOverhead) \times n\). Furthermore, each iteration in the loop creates the space that is needed for the object header, as previously illustrated in Figure 4.6.

**Complexity**

The complexity for the modified `MCentral_Grow` is defined by the old complexity added up with the complexity of the modifications (4.8).

\[
O(MCentral\_Grow_{\text{new}}) = O(MCentral\_Grow_{\text{old}}) + O(mcentral\_modications) \quad (4.8)
\]

The modifications include pointer dereferences and a loop containing assignments. The iterations of the loop are bounded by the number of size class blocks in a maximum number of pages, both defined at compile-time. The complexity of `mcentral\_modications` is thus constant (4.9).

\[
O(mcentral\_modications) = O(1) + \text{constant}\_n\_iterations \times O(1) = O(1) \quad (4.9)
\]
4. Short-term Memory Implementation in Go

The overall complexity of $MCentral_{Grow_{new}}$ has not changed as can be seen in (4.10). Again, a constant runtime overhead is added.

$$O(MCentral_{Grow_{new}}) = O(MCentral_{Grow_{old}}) + O(1) \quad (4.10)$$

4.5.3. Allocator Modifications – scm_free

In order to deallocate objects, a SCM variant of a free method has been created. This is necessary since the Go GC can process whole pages of objects (of a size class) in a batch, looking up the specific references only once. After this lookup, the GC can re-use certain flags, one of those being the garbage collector flag. Since looking up this kind of information is costly (although still constant), the necessary header fields have been created. By using these fields, the scalability can be maintained while still only using one word overhead per object (for small objects). Listing 4.6 shows a simplified version of this free function. Before being able to free an object, it is necessary to know its size(class).

As Figure 4.5 illustrates, the size class field is at the same position for both, so this can be read in any case. Although in the following the distinction between large and small objects is made, the principle stays the same. Instead of looking up the span (for large objects) or the garbage collector reference flag (for small ones) through the $MSpan$-tree, they are calculated by evaluating the header fields. Finally, the objects are freed using their corresponding free functions from the runtime.

Listing 4.6: Schematic scm_free

```c
void scm_free(void *object) {
    if (object == nil)
        return;

    sizeclass = OBJECT_HEADER(object)->sizeclass;

    if (sizeclass == 0) { // Large object
        s = OBJECT_HEADER_LARGE(object)->span;
        s->gcref0 = RefFree;
        MHeap_Free(&mheap, s, 1);
    } else { // Small object
        refsmall = (uint32*)((uintptr)object +
            OBJECT_HEADER(object)->gcref_offset);
        *refsmall = RefFree;
        c = m->mcache;
    }
}
```
4. Short-term Memory Implementation in Go

```go
size = class_to_size[sizeclass];
MCache_Free(c, object, sizeclass, size);
}
}
```

**Complexity**

Like in the malloc modifications the complexity can be different in both paths (4.11).

\[
\mathcal{O}(\text{scm\_free}) = \begin{cases} 
\mathcal{O}(\text{modifications}_{\text{large}}) + \mathcal{O}(\text{MHeap\_Free}) & \text{size\_class} = 0 \\
\mathcal{O}(\text{modifications}_{\text{small}}) + \mathcal{O}(\text{MCache\_Free}) & \text{size\_class} > 0
\end{cases}
\]  

(4.11)

However, it is clear that the added modifications do belong to the constant complexity class, since they only contain assignments and dereferences. As a result, the overall complexity of \text{scm\_free} can be noted as shown in (4.12) and (4.13).

\[
\mathcal{O}(\text{scm\_free}) = \begin{cases} 
\mathcal{O}(1) + \mathcal{O}(\text{MHeap\_Free}) & \text{size\_class} = 0 \\
\mathcal{O}(1) + \mathcal{O}(\text{MCache\_Free}) & \text{size\_class} > 0
\end{cases}
\]  

(4.12)

\[
\mathcal{O}(\text{scm\_free}) = \begin{cases} 
\mathcal{O}(\text{MHeap\_Free}) & \text{size\_class} = 0 \\
\mathcal{O}(\text{MCache\_Free}) & \text{size\_class} > 0
\end{cases}
\]  

(4.13)

Like in previous examples, the complexity of the overall algorithm, i.e. free, is not changed.

### 4.5.4. StmTick

The tick operation follows the semantic model of STM, as described in Section 3. Listing 4.7 shows a simplified version of the tick operation. While it is fully functional, it lacks error handling, such as checking the DescriptorRoot against nil. This may be the case if no refresh happened before a tick.

**Listing 4.7: Schematic StmTick**

```go
func StmTick() {
    DescriptorRoot *root;
    root = m->curg->root;
    root->current_local = (root->current_local + 1)
```
The implementation sets the new goroutine local time modulo a maximum expiration extension that is fixed at compile-time. This fix ensures that the buffer is bounded. After incrementing the local time, the old descriptors (and thus objects) are expired using Expire_Local. Finally, the function tries to reclaim one object using Expire_If_Exists. Before going into the details about the complexity, it is important to investigate the two used functions further.

```c
void Expire_Local(DescriptorRoot *root) {
    to_be_expired_index = root->current_local - 1;
    if (to_be_expired_index < 0)
        to_be_expired_index += SCM_MAX_EXPIRATION_EXTENSION + 1;

    if (root->first_local[to_be_expired_index] != nil &&
        root->first_local[to_be_expired_index]->number_of_descriptors != 0){
        concat_local_to_expired(root, to_be_expired_index);
        root->first_local[to_be_expired_index] = DescriptorPage_New(root);
        root->last_local[to_be_expired_index] = root->first_local
            [to_be_expired_index];
    }
}
```

Listing 4.8 shows the once more simplified code that is used to expire the list of objects of the last time instance. The algorithm does not free the objects, but appends them to a list of to-be-freed objects that is then iteratively processed. The function DescriptorPage_New is skipped because it only gets a new DescriptorPage from either a pool or it allocates one and sets its internals.

One free iteration process happens directly after expiring the local list, in the function Expire_If_Exists. Listing 4.9 shows the implementation that checks whether an object is to be expired. First, if possible, the function obtains an expired object. Secondly, the descriptor-counter is atomically decremented and checked against zero. If this check
succeeds, the object is ready to be freed by a modified allocator.

Listing 4.9: Schematic Expire_If_Exists

```c
void Expire_If_Exists(DescriptorRoot *root) {
    expired_object = Get_Expired_Object(root);
    if (expired_object != nil) {
        if ((atomic_int32_dec_test(
            (int32*)(&expired_object->descriptor_counter))) {
            object = (void*) PAYLOAD_OFFSET(expired_object);
            scm_free(object);
        }
    }
}
```

Finally, before analyzing the complexity of the solution, Listing 4.11 shows the assembly code for the atomic `atomic_int32_dec_test` function. Note that “decrement and test” is being rewritten with “exchange and add”, as listed below in Listing 4.10.

Listing 4.10: Decrement-and-test → exchange-and-add

```c
#define atomic_int32_dec_test(atomic) \
    (atomic_int32_exchange_add \left( atomic, -1) == 1)
```

Listing 4.11: `atomic_int32_exchange_add`

```c
// int32 atomic_int32_exchange_add(int32 *atomic, int32 val)
TEXT atomic_int32_exchange_add(SB), 7, $0
    MOVQ 8(SP), AX // first parameter: int32 *atomic
    // (pointer to 32 bit=> 64bit)
    MOVL 16(SP), BX // second parameter: int32 val (32 bit value)
    LOCK
    XADDL BX, (AX) // exchange and add using inverse intel syntax
    // Intel: XADD r32/m32, r32
    MOVL BX, AX // Assume return value is AX
    RET
```

**Complexity**

For the sake of simplicity the above description does not contain all sub-functions. This complexity analysis assumes that the non-described functions are constant, i.e. $O(1)$,
which they are. The complexity of \( \text{StmTick} \) can be calculated by (4.14).

\[
\mathcal{O}(\text{StmTick}) = \mathcal{O}(\text{Deref\_and\_Assignment}) + \mathcal{O}(\text{Expire\_Local}) + \mathcal{O}(\text{Expire\_If\_Exists})
\]

(4.14)

Of course, \( \mathcal{O}(\text{Deref\_and\_Assignment}) = \mathcal{O}(1) \), but the other functions need to be further investigated.

\[
\mathcal{O}(\text{Expire\_Local}) = \mathcal{O}(\text{concat}) + \mathcal{O}(\text{DescriptorPage\_New}) = \mathcal{O}(1) + \mathcal{O}(1) = \mathcal{O}(1)
\]

(4.15)

As already mentioned, the complexity for a concatenation with known first and last pointer is constant. The creation of a \text{DescriptorPage} is also constant and thus the complexity of \text{Expire\_Local} is constant as well (4.15). The complexity of \text{Expire\_If\_Exists} can be determined by investigating the function as shown in (4.16) and (4.17).

\[
\mathcal{O}(\text{Expire\_If\_Exists}) = \begin{cases} 
\mathcal{O}(1) & \text{object} = \text{nil} \lor \text{descriptorcounter} > 1 \\
\mathcal{O}(\text{scm\_free}) & \text{otherwise}
\end{cases}
\]

(4.16)

\[
\mathcal{O}(\text{Expire\_If\_Exists}) = \mathcal{O}(\text{scm\_free}) = \begin{cases} 
\mathcal{O}(\text{MHeap\_Free}) & \text{size\_class} = 0 \\
\mathcal{O}(\text{MCache\_Free}) & \text{size\_class} > 0
\end{cases}
\]

(4.17)

Since the semantics of \text{Expire\_If\_Exists} is that it should free one object if possible, the resulting complexity aligns to the one of \text{scm\_free}. This approach is called incremental collection (4.18).

\[
\mathcal{O}(\text{Incremental\_Collection}) = \mathcal{O}(\text{Expire\_If\_Exists})
\]

(4.18)

It is of course possible to alter this behaviour by not only freeing one object at each operation, but cleaning up the whole expired list upon each API call, called eager collection. As a result, the complexity for eager collection changes to being dependent on all object
4. Short-term Memory Implementation in Go

in the expiration list (4.19).

\[ O(Eager\_Collection) = n \times O(\text{expire}_If\_Exists), \quad n = \text{number of object in expired list} \]

(4.19)

4.5.5. StmRefresh

The last operation that is needed to satisfy the STM API is a refresh function. The function itself is split into two parts. The first is a Go part that traps the highest level call. This function then uses an inner internal refreshing call, written in C, passing the needed parameters. The splitting is needed because the internal refreshing is called recursively for known Go data types, i.e. slices. Listing 4.12 shows both functions.

Listing 4.12: StmRefresh and internal_refresh

```go
func StmRefresh(obj Eface, ext int32) {
    internal_refresh(obj.type, obj.data, ext);
}

void internal_refresh(Type* t, byte* data, int32 ext) {
    if (m->curg->root == nil) {
        m->curg->root = scm_register_g();
    }
    m->curg->root->do_gc = 0;
    root = m->curg->root;
    od = OBJECT_HEADER(data);

    if (od->descriptor_counter == 0) {
        *(uint32*)((uintptr)data+od->gref_offset) = RefFree;
    }
    atomic_int32_exchange_add((int32*)&od->descriptor_counter, 1);

    Expire_if_Exists(root);
    Descriptor_Insert_Local(OBJECT_HEADER(data), root, ext);

    // Slice handling; recursively refreshing all data fields
    if (t->kind == KindSlice) {
        internal_refresh(((SliceType*)t)->elem, ((Slice*)data)->array, ext);
    }
}
```
4. Short-term Memory Implementation in Go

The internal function can be divided into several different parts. The first part deals with the registration of the thread-local data structures, that is, a struct containing the ring buffer for expiration lists and an expired list. The function is not further explained as it just allocates a struct and stores it in the current goroutine’s scope.

The second part sets the do_gc flag to create a pattern that avoids GC runs, as explained in Section 4.4.2.

Finally, the third part handles the actual refreshing of the object. This means that it sets the garbage collector flag to RefFree if the object is first encountered in a STM context. The GC then ignores the object as it is already considered as free. Furthermore, it has not yet been returned to any free list and is thus also not available for the allocator. The descriptor is then inserted into the DescriptorPages after one object has, possibly, been collected. The function is triggered recursively if a slice is encountered to ensure that also the actual slice data is refreshed properly (see Section 4.2.3 for memory layout).

 Complexity

The complexity of the StmRefresh method can be summarized by (4.20). Since it only refers to an internal refreshing method, the complexity is specified by this internal refreshing.

\[ O(StmRefresh) = O(internal\_refresh) \]  \hspace{1cm} (4.20)

The overall complexity of the internal_refresh function is illustrated by (4.21). It is summarized by the single tasks in the function. While \( O(Expire\_If\_Exists) = O(1) \) has already been shown before, we need to specify the rest of the terms.

\[
O(internal\_refresh) = n \ast (O(scm\_register\_g) + \\
O(exchange) + O(Expire\_If\_Exists) + \\
O(Descriptor\_Insert\_Local)),
\]

\[ n = \text{number of slice recursions} \]  \hspace{1cm} (4.21)

The function scm_register_g registers a new constant-size goroutine-local buffer in the context of the running goroutine. This is needed when a refresh happens in a function for the first time. (StmTick checks against a nil goroutine-local buffer and returns immediately.) Furthermore, the local buffer does not need to be created as long as one is in a global pool. Only if the global pool is also empty, e.g. this is the first goroutine
using STM, a new buffer has to be allocated through the Go internal allocation function, FixAlloc_Alloc (4.22).

\[
O(scm\_register\_g) = \begin{cases} 
O(FixAlloc\_Alloc), & \text{first refresh} \\
O(1), & \text{otherwise}
\end{cases}, \text{first refresh in a goroutine}
\]

\[
O(\text{Desciptor\_Insert\_Local}) = O(1)
\]

The atomic decrement of the integer counter of descriptors is obviously constant, thus \(O(\text{exchange}) = O(1)\). See Listing 4.11 for the code.

Finally, without going into too much detail, \(O(\text{Desciptor\_Insert\_Local}) = O(1)\). This is the case because the creation of a descriptor is only an insert into an already existing allocation page.

The overall result of the complexity can be seen in (4.24). If the object is not a slice, then \(n = 1\), making it a constant operation. Certainly this only applies after the first refresh happened, as this first refresh could in the worst case allocate the buffers that are needed by STM (in a internal runtime section).

\[
O(\text{StmRefresh}) = O(n), n = \text{number of slice hierachies}
\]

4.6. Complexity Summary

Table 4.1 once more summarizes the time complexity of the implemented functions and modifications of the runtime. The modifications made to the internal malloc function of the Go runtime are constant in both size class cases. The setup mechanisms (preparing the internal header) added to the MCentral structure are bounded by a loop-constant \(n\), which is known at compile-time. The internal free function is dependent on the underlying allocator (MHeap_free and MCache_Free) and adds only constant overhead. The internal refresh considers the internal slice structure and thus has linear complexity bounded
by the number of slices recursions. The incremental ticking function has constant com-
plexity while the eager version is linearly dependent on the number of objects that have 
expired. The externally visible function for refreshing objects just utilizes the internal one 
and therefore falls in the same complexity class.

Overall the changes only add constant overhead to the existing allocator with two ex-
ceptions, recursive slices and eager collection runs, which both result in linearly bounded 
runtimes.

<table>
<thead>
<tr>
<th>Function/Modification</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>mallocgc modifications</td>
<td>( O(1) ) ( \leq ) MaxSmallSize ( O(1) ) size &gt; MaxSmallSize</td>
</tr>
<tr>
<td>mcentral modifications</td>
<td>( O(1) + n \cdot O(1), n = ) number of size class blocks</td>
</tr>
<tr>
<td>scm_free</td>
<td>( O(MHeap_Free) ) size_class = 0 ( O(MCache_Free) ) size_class &gt; 0</td>
</tr>
<tr>
<td>internal_refresh</td>
<td>( O(1) \cdot n, n = ) number of slice recursions</td>
</tr>
<tr>
<td>StmTick (Incremental)</td>
<td>( O(scm_free) )</td>
</tr>
<tr>
<td>StmTick (Eager)</td>
<td>( O(scm_free) \cdot n, n=)number of expired objects</td>
</tr>
<tr>
<td>StmRefresh</td>
<td>( O(internal_refresh) )</td>
</tr>
</tbody>
</table>

Table 4.1.: Summary of function/modification complexities

4.7. Example

This last implementation section now deals with a trivial example to show how all the 
previously mentioned algorithms and buffers work together to create the SCM for STM.

Listing 4.13: Go STM example

```go
package main

import (  
    "runtime"
)

type ImmortalData struct {  
    information int
}  
```
4. Short-term Memory Implementation in Go

```go
10 type IterationData struct {
11   other_information int
12 }
13
15 func main() {
16   var tmp_data *IterationData
17   
18   global_data := new(ImmortalData)
19   
20   for i:=0; i < getMaxDepth; i++ {
21     tmp_data = new(IterationData)
22     runtime.StmRefresh0(tmp_data)
23     // fill tmp_data with some function
24     // update global_data with information from tmp_data
25     runtime.StmTick()
26   }
27   // do something with global_data
28 }
```

Listing 4.13 shows an idiomatic Go program using STM. The following paragraph now summarizes what happens regarding the state of the STM internals:

1. The program’s entry point the `main` function, starts by getting some global data, `global_data`, that needs to survive until the end of the program execution. Nothing regarding STM happens until entering the loop, as the backwards compatible approach does not introduce any overhead.

2. The program enters the loop and stores temporary data in `tmp_data`. Still nothing regarding STM happened.

3. The function `StmRefresh0` is used to transfer the function to using STM by incrementing the counter of descriptors by one and setting its GC flag to free, i.e. handled by STM. Before this happens, a new goroutine-local storage buffer is allocated and stored in the goroutine-local context.

4. Global data is updated using temporary data.

5. `StmTick` increments the goroutine-local time and expires everything of the current time, including the temporary data.
6. The next iteration-step re-acquires temporary data like in (3), but uses the already existing goroutine-local buffer when creating a descriptor.

7. Global data is updated using temporary data.

8. StmTick increments the goroutine-local time and expires everything of the current time, including the temporary data.

9. The loop is continued at (6) until the maximum depth is reached.

10. The program performs operations on the global data. The goroutine-local buffer is destroyed upon return of the function as it is the main function. Any other goroutine would have returned the goroutine-local buffer to a global pool.

Note that the loop is self-compacting as at any STM operation one object is freed and directly reused by the following allocation operation. Thus the live heap-size stays the same while the loop is being executed.
5. Experiments

This chapter deals with experiments and benchmarks that have been taken to provide an overview of the capabilities of the STM enabled Go runtime. Therefore we compare the standard Go runtime memory manager, i.e. a stop-the-world mark-sweep GC, with the hybrid SCM enabled runtime. The capabilities are recorded in time and space dimension for two different benchmarks. The first benchmark is a micro measurement focusing on performance. This means that a collector competing in this benchmark has to provide high throughput, low memory usage and preferably low latency. The second benchmark is macro-oriented although it does not provide a real application. It is a webserver distributing static webpages. This benchmark includes measurements of space and latency as response time as these are important for the webhost and consuming client.

5.1. Test Environment

A single platform is used to acquire the measurements for both benchmarks. The characteristics of the machine are listed in Table 5.1. The benchmarks all use in-program measuring of time and space. As a result, the programs are independent from any start-up routines of the underlying operating system. We tried to ensure that no other process interrupted the benchmarks through operating system scheduling, by running the benchmarks with a lower nice value than the default one. The privileging of the benchmarks cannot be

\[\text{http://linux.die.net/man/3/nice}\]

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>2x AMD Opteron DualCore, 2.0 GHz</td>
</tr>
<tr>
<td>RAM</td>
<td>4 GB</td>
</tr>
<tr>
<td>OS</td>
<td>Linux 2.6.32-31-generic (Ubuntu)</td>
</tr>
<tr>
<td>Go compiler/runtime</td>
<td>6g, release 2010-11-02</td>
</tr>
</tbody>
</table>

Table 5.1.: System configuration
guaranteed, however, as the underlying operating system has been Linux, which comes per default without any real-time or scheduling patches.

5.2. Single-Threaded – Tree Benchmark

The benchmark presented here is a variant of a tree construction benchmark of The Computer Language Benchmarks Game\(^2\). The complete source code of Go program can be found in the Appendix B.

The benchmark is used to test the validity of a garbage collector and the performance regarding throughput. Validity can be shown whether the program crashes due to lack of available heap memory, while the throughput is measured by the time that has been needed to complete the benchmark. The same characteristics apply to the benchmark when testing it with STM.

**Characteristics**

The benchmark creates various different-sized trees using a single internal node structure, that not only holds the left and right children, but also an internal integer value. Furthermore, a validation function is used to check the internal value of the items, i.e. summing up all values. The program is started using an input parameter \(n\), defining the maximum size of the trees that are created. After initialization, a long-lived tree with a maximum depth is created. This allocated tree has to survive till the end of the program. After the long-lived tree has been created, a loop creates various short-lived trees. The loop starts with creating trees of the minimum depth (4) and increases the depth until \(n\) is reached. The root reference to these short trees is only stored in a temporary variable, making them collectable after each iteration.

**Transformation to Short-term Memory**

The transformation of the tree benchmark to STM also shows the ease-of-use when converting existing programs.

As a first step, a tick call is placed after every short-lived tree creation. Due to the nature of the code that recursively creates the tree, an additional function for node-creation,

\(^2\)http://shootout.alioth.debian.org/
which also includes a refresh call, needs to be implemented. Since there exists a long-lived tree, that is not subject of STM, the original function remains in the code. Listing 5.1 shows the essence of the transformation. A second step then disables the GC completely because we, as programmers, know that the long-lived tree is actually an immortal tree that stays until program termination and therefore no GC is needed at all.

```
Listing 5.1: Go tree benchmark transformation

// original
return &Node{ item , bottomUpTreeStm(2*item−1, depth−1),
    bottomUpTreeStm(2*item, depth−1)}

// transformed
n = &Node{ item , bottomUpTreeStm(2*item−1, depth−1),
    bottomUpTreeStm(2*item, depth−1)}
runtime.StmRefresh0(n)
return n

// original
check += bottomUpTreeStm(i , depth ).itemCheck()
check += bottomUpTreeStm(-i , depth ).itemCheck()

// transformed
check += bottomUpTreeStm(i , depth ).itemCheck()
runtime.StmTick()
check += bottomUpTreeStm(-i , depth ).itemCheck()
runtime.StmTick()
```

5.2.1. Results

As already mentioned, the programs were modified furthermore to allow the measurements to be taken in the program. For time benchmarks requiring timing information, there has been a print added at each position a timestamp is needed. For space oriented views a print to output heap information has been added in the main program loop.

Before going into detail about the results, it is important to note the different modifications of the program that have finally been used to fetch the measurements:
5. Experiments

- **Immortal**: No garbage collection or STM is used at all. All memory is immortal.

- **GC**: The standard Go GC is used. Besides measuring modifications this benchmark uses the original program.

- **STM+GC**: As described in the transformation, this configuration uses STM for short-lived trees, but keeps the GC enabled to handle everything else.

- **STM only**: The GC is also disabled because the only long-lived object is a permanent object.

**Time**

Figure 5.1 shows the overall throughput of the systems by measuring the overall time the program needed for execution.

![Figure 5.1.: Tree benchmark overall execution time](image)

The benchmark has been executed 15 times per configuration and shows the average value of the overall time that has been needed to execute the benchmark (neglecting the program start). The red bar (Immortal) shows the ideal time that would’ve been needed...
5. Experiments

<table>
<thead>
<tr>
<th>n</th>
<th>immortal</th>
<th>GC</th>
<th>STM+GC</th>
<th>STM only</th>
<th>P(STM+GC)</th>
<th>P(STM only)</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>3.875</td>
<td>6.526</td>
<td>5.046</td>
<td>4.724</td>
<td>22.679%</td>
<td>27.613%</td>
</tr>
<tr>
<td>16</td>
<td>9.163</td>
<td>15.420</td>
<td>11.842</td>
<td>11.115</td>
<td>23.204%</td>
<td>27.918%</td>
</tr>
<tr>
<td>17</td>
<td>18.439</td>
<td>31.183</td>
<td>23.820</td>
<td>22.339</td>
<td>23.612%</td>
<td>24.279%</td>
</tr>
</tbody>
</table>

Table 5.2.: Overall time [s] – tree benchmark

If deallocation could happen in zero time. The green bar (GC) shows the time that has been needed for the standard Go system. The blue (STM+GC) and grey (STM only) show the STM enabled versions. Table 5.2 summarizes the results once more. The last two columns show the performance increase \( P[\%] = 100 - \frac{\text{modification time}}{\text{GC time}} \times 100 \). The average increase for STM+GC over the maximum tree depth parameter \( n \) is 23.165%. For STM only the average increase is 26.603%. Of course “STM only” is faster, as the GC is completely disabled. Since the GC cannot be disabled in a more general use-case, the benchmark is negligible.

Space

Figure 5.2 shows the performance of the systems in the space dimension. The STM pagesize was fixed to 32 byte and the benchmark was run with \( n = 9 \). The immortal modification has been removed from this benchmark as the whole program is a memory leak. In order to get an aligned picture for all modifications, the heap statistics have not been collected over time, but over allocator operations.

What can be noticed is that the standard GC performs as expected, leaving garbage behind until a mark-sweep run is triggered and cleans everything up. When looking at the STM implementations, one can see the allocations of the overhead in the difference of the ascent between \( 0 - 5000 \) allocations. The memory consumption then stays the same as at each refresh (which happens in the tree-creation) and an internal node is freed right after one has been allocated. The following behavior therefore illustrates how self-compaction can be achieved using STM. Programs that may have such a property are also described in Chapter 6.
5. Experiments

Figure 5.2.: Tree benchmark overall live heap size

Latency

Figure 5.3 illustrates the latency measurements of the tree benchmark. One can notice that the peaks for the mark-sweep runs of the GC are aligned with the ones of Figure 5.2. The figure also includes the measurements for the immortal modification as it shows the noise of the rest of the memory management operations.

The latency properties are summarized in Table 5.3. The arithmetic mean \( \mu \) and standard deviation \( \sigma \) are calculated as shown in (5.1) and (5.2), respectively. \( N \) is the number of measurements and \( x_i \) the \( i^{th} \) measurement.

\[
\mu = \frac{1}{N} \sum_{i=1}^{N} x_i \quad (5.1)
\]

\[
\sigma = \sqrt{\frac{1}{N} \sum_{i=1}^{N} (x_i - \mu)^2} \quad (5.2)
\]

Table 5.3 shows that the average latency stays the same, while STM+GC and STM only
5. Experiments

have a standard deviation that is within the dimension of the immortal modification. The GC modification has a standard deviation that is 10 times higher than the immortal modification. The same trend is illustrated by the maximum values, with STM+GC suffering from the impact of the GC.

<table>
<thead>
<tr>
<th></th>
<th>μ [ms]</th>
<th>σ [ms]</th>
<th>max [ms]</th>
</tr>
</thead>
<tbody>
<tr>
<td>immortal</td>
<td>8.199</td>
<td>1.144</td>
<td>68</td>
</tr>
<tr>
<td>GC</td>
<td>8.254</td>
<td>11.435</td>
<td>868</td>
</tr>
<tr>
<td>STM+GC</td>
<td>8.502</td>
<td>1.234</td>
<td>284</td>
</tr>
<tr>
<td>STM only</td>
<td>8.228</td>
<td>1.223</td>
<td>77</td>
</tr>
</tbody>
</table>

Table 5.3.: Tree benchmark latency summary

5.3. Multi-Threaded – A Webserver

Go developers forgive me the term “threaded” in the headline as the execution actually happens in goroutines. Nevertheless, the term is more convenient when talking about
having concurrent threads of execution.

The benchmark here illustrates the capabilities of STM in a macro-oriented benchmark than the tree-creation, i.e. it shows a webserver use-case that is also illustrated as an example on the Go documentation pages.

The source code of the benchmark can be found in Appendix C. The program serves a static webpage on http://localhost:8080/google, hosting the Google main page. In order to obtain the measurements, the performance tool httperf has been used. The webserver has been tested under a load of 600 connections per second with an overall of 1200 connections and a timeout of 5 seconds.

Note that the first attempt did involve rewriting the Go documentation webserver, but has been found unsuitable, as the documentation server also includes an indexer that runs in the background. However, a more substantial benchmark that may involve the Go documentation server is subject of future investigations.

Figure 5.4.: Webserver live heap-size

3http://golang.org/doc/codelab/wiki/
4http://www.google.com
5http://www.hpl.hp.com/research/linux/httperf/
5. Experiments

5.3.1. Results

The results are shown in the dimensions space and latency, as an overall timing behavior cannot be measured on a program like a webserver. The throughput of the webserver is implicitly measured by the latency it provides.

![Webserver latency measurements](image)

**Figure 5.5.: Webserver latency measurements**

**Space**

Figure 5.4 shows the live heap size of the webserver while serving the requests. The live heap size has been measured by printing heap statistics while the webserver has been running. It can be noticed that in contrast to the tree-creation benchmark this benchmark shows how a mix of a GC and STM results in a garbage-collected behavior with a slight increase of memory utilization. Since only the static body is stored in an STM object (see Appendix C) the self-compacting behavior is not significant.
5. Experiments

<table>
<thead>
<tr>
<th>Latency Summary</th>
<th>μ [ms]</th>
<th>σ [ms]</th>
<th>max [ms]</th>
</tr>
</thead>
<tbody>
<tr>
<td>GC</td>
<td>1.950</td>
<td>1.808</td>
<td>16.208</td>
</tr>
<tr>
<td>STM+GC</td>
<td>1.427</td>
<td>0.559</td>
<td>9.308</td>
</tr>
</tbody>
</table>

Table 5.4.: Webserver benchmark latency summary

Latency

Figure 5.5 illustrates the latency behavior of the webserver while serving the request of httpperf. In contrast to previous measurements, these illustrate the behaviour experienced by httpperf, the webserver profiler.

Table 5.4 once more summarizes the measurements, using the arithmetic mean and standard deviation from (5.1) and (5.2). Once again it can be noticed that while the mean is still in the same domain, the standard deviation is significantly lower making the system more responsive overall.
6. Conclusion

In this thesis we have introduced a new memory management system for the programming language Go. The system is based on the so-called short-term memory model. Traditional explicit or implicit heap management approaches, described in Chapter 2, are based on a persistent memory model in which objects are allocated for an unknown amount of time. In contrast to this, the STM model uses expiration dates to mark objects as live for a certain amount of time. These fundamental differences result in a different point of view regarding objects that are allocated and deallocated in programs. In the traditional model the programmer or underlying system has to care about objects that need to be freed, i.e. not-needed objects. Using STM, the programmer needs to focus on the objects that are important when continuing the computation, the i.e. needed objects. Although there exist no strict notion for the term usability, we claim that this different point of view, which is explained in Chapter 3, may provide a system that is easier to use.

The implementation, which is described in Chapter 4, uses SCM to achieve in-band collection of objects that are expired. The modified runtime is also completely backwards compatible, meaning that existing Go programs can run without any modifications. The transferring of objects to an STM managed environment happens by a constant-time library call issued on the object that needs to be transferred. As all library code, that requires synchronization, is executed in the Go scheduler, which only schedules a thread at a time, no new locks are introduced. As a result the system is scalable in a multi-threaded environment. Furthermore, we have shown that all operations add only constant time overhead to the existing runtime implementation, making the collection incremental. In addition to an incremental collection it is also possible to collect all expired objects at once, minimizing the memory footprint.

The experiments and benchmarks that are shown in Chapter 5 back up these claims by showing that the SCM implementation in Go is backwards-compatible to the Go baseline runtime and outperforms the existing GC in time (throughput), space consumption (constant) and latency (less jitter). A micro benchmark tested these capabilities in all di-
6. Conclusion

In conclusion, the benchmark showed that STM used with the existing GC can increase the timing performance by up to 23.612%. In contrast to the baseline system where the memory consumption was alternating between 0.15MB and 0.3MB, the memory consumption using STM was constant at about 0.24MB. Latency-wise, we showed that STM could not improve the average pause time, but reduce the standard deviation by 89%. The maximum pause time was also reduced from 824ms to 284ms. In the webserver benchmark, we showed that the average response time could be improved by about 27%. The standard deviation could be decreased by about 60%.

The performance increase and system that may be easier to handle come at the price of losing safety that a garbage collected system provides.

6.1. Usage – Well Suited Applications

STM can be used in any system that relies on short-lived objects. A system that would highlight the capabilities of the SCM approach acquires memory at program startup and then enters a loop that needs to acquire short-living objects in each iteration. Such an application would not only provide better latency with STM but also use less memory compared to a GC as it is inherently self-compacting, relying on the previously freed slots when acquiring new objects.

6.2. Future Work

There are several possibilities to further increase the usability of the system. Fully usable global-time management still needs to be implemented. Furthermore, a real macro benchmark could be used to show the full capabilities of STM.

Another topic that needs to be discussed is the usage of STM which is assisted by a compiler and possibly memory traces that have been taken at runtime. It may be possible to (over-)approximate the expiration dates of objects using traces that have been previously acquired when actually using the application.
Bibliography


74


A. Abbreviations

API  Application Programmable Interface
CPU  central processing unit
GC   garbage collector
GCC  GNU Compiler Collection
GNU  GNU is not Unix
LIFO last-in first-out
MM   memory management
NPTL Native POSIX Thread Library
SCM  self-collecting mutators
STM  short-term memory
B. Computer language benchmarks
game – tree benchmark

This section holds the source code of the tree benchmark program that is included in the Go sources with the tag release.2010-11-02.

/*
Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

* Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

* Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

* Neither the name of "The Computer Language Benchmarks Game" nor the name of "The Computer Language Shootout Benchmarks" nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.*/

77
package main

import (
    "flag"
    "fmt"
)

var n = flag.Int("n", 15, "depth")

type Node struct {
    item int
    left, right *Node
}

func bottomUpTree(item, depth int) *Node {
    if depth <= 0 {
        return &Node{item: item}
    }
    return &Node{item, bottomUpTree(2*item-1, depth-1),
        bottomUpTree(2*item, depth-1)}
}

func (n *Node) itemCheck() int {
    if n.left == nil {
        return n.item
    }
    return n.item + n.left.itemCheck() - n.right.itemCheck()
}

const minDepth = 4
B. Computer language benchmarks game – tree benchmark

```go
func main() {
    flag.Parse()

    maxDepth := *n
    if minDepth+2 > *n {
        maxDepth = minDepth + 2
    }
    stretchDepth := maxDepth + 1

    check := bottomUpTree(0, stretchDepth).itemCheck()
    fmt.Printf("stretch_tree_of_depth_%d\t_check:%d\n", stretchDepth, check)

    longLivedTree := bottomUpTree(0, maxDepth)

    for depth := minDepth; depth <= maxDepth; depth += 2 {
        iterations := 1 << uint(maxDepth−depth+minDepth)
        check = 0

        for i := 1; i <= iterations; i++ {
            check += bottomUpTree(i, depth).itemCheck()
            check += bottomUpTree(−i, depth).itemCheck()
        }
        fmt.Printf("%d\t_trees_of_depth_%d\t_check:%d\n", iterations*2, depth, check)
    }
    fmt.Printf("long_lived_tree_of_depth_%d\t_check:%d\n", maxDepth, longLivedTree.itemCheck())
}
```
C. Webserver benchmark

This section contains the source code of the webserver benchmarks for the unmodified garbage collector version and the STM enabled program in the Listings C.1 and C.2, respectively.

Listing C.1: Webserver using the Go GC

```go
package main

import (
    "fmt"
    "http"
    "io/ioutil"
    "runtime"
)

func handler(w http.ResponseWriter, r *http.Request) {
    body, _ := ioutil.ReadFile("Google.html")
    fmt.Fprintf(w, string(body))
    if body == nil {
        
    }
}

func main() {
    // benchmark setup, disable Go profiling
    runtime.MemProfileRate = 0

    http.HandleFunc("/google", handler)
    http.ListenAndServe(":8080", nil)
}
```

Listing C.2: Webserver using STM and the GC

```go
//
```
package main

import (
    "fmt"
    "http"
    "io/ioutil"
    "runtime"
)

func handler(w http.ResponseWriter, r *http.Request) {
    body, _ := ioutil.ReadFile("Google.html")
    runtime.StmRefresh0(body)
    fmt.Fprintf(w, string(body))
    runtime.StmTick()
}

func main() {
    // benchmark setup, disable profiling
    runtime.MemProfileRate = 0

    http.HandleFunc("/google", handler)
    http.ListenAndServe(":8080", nil)
}