Automatic Memory Management Techniques for the Go Programming Language

Matthew Ryan Davis

Submitted in total fulfilment of the requirements of the degree of Doctor of Philosophy

September 2015
Department of Computer Science and Software Engineering
The University of Melbourne

Produced on archival quality paper.
Memory management is a complicated task. Many programming languages expose such complexities directly to the programmer. For instance, languages such as C or C++ require the programmer to explicitly allocate and reclaim dynamic memory. This opens the doors for many software bugs (e.g., memory leaks and null pointer dereferences) which can cause a program to crash. Automated techniques of memory management were introduced to relieve programmers from managing such complicated aspects. Two automated techniques are garbage collection and region-based memory management. The more common technique, garbage collection, is primarily driven by a runtime analysis (e.g., scanning live memory and reclaiming the bits that are no longer reachable from the program), where as the less common region-based technique performs a static analysis during compilation and determines program points where the compiler can insert memory reclaim operations. Each option has its drawbacks. In the case of garbage collection it can be computationally expensive to scan memory at runtime, often requiring the program to halt execution during this stage. In contrast, region-based methods often require objects to remain resident in memory longer than garbage collection, resulting in a less than optimal use of a system's resources.

This thesis investigates the less common form of automated memory management (region-based) within the context of the relatively new concurrent language Go. We also investigate combining both techniques, in a new way, with hopes of achieving the benefits of a combined system without the drawbacks that each automated technique provides alone. We conclude this work by applying our region-based system to a concurrent processing environment.
Declaration

This is to certify that:

• the thesis comprises only my original work towards the PhD except where indicated in the Preface,

• due acknowledgement has been made in the text to all other material used,

• the thesis is fewer than 100,000 words in length, exclusive of tables, maps, bibliographies and appendices.

Matthew Ryan Davis
This thesis is the result of over three years of work and collaboration with some of the brightest people I have ever worked with, my supervisors. I have benefited greatly from our weekly discussions, impromptu brainstorming sessions, and our collaborative paper writing process.

Chapter 3 and Chapter 5 are based on publications [17, 18] written together with my supervisors. I am listed as the primary author on both of these works, having done the implementation, experimentation, and original drafts. These two chapters derive from our publications to the ACM SIGPLAN Workshop on Memory Systems Performance and Correctness 2012 and 2013 respectively. Chapter 6 extends a discussion that was started in Chapter 3 [17].
I would like to thank my supervisors: Peter Schachte, Zoltan Somogyi, and Harald Sondergaard for all of their hard work, genius ideas, brain storming sessions, and impeccable pedantic reasoning towards correctness. I have never worked with such a smart team in my life. I would also like to thank the University for its great bandwidth and procurement of freeze-dried caffeine that kept me trudging through the hours of finger-physical labor that this degree required. This research process has been an incredibly rewarding experience, and I have learned quite a lot through this collaborative effort. Not only have I broadened my knowledge on compilers and memory management, but I believe that I have also become a stronger developer and critical thinker.

I was awarded both scholarship funding from NICTA and the University, and am forever grateful for their financial support. I would also like to thank my good friend, Julien Ridoux, for his friendship. Julien allowed me to work part-time with his and Darryl Veitch’s team researching high precision BSD/Linux kernel and network timing as well as network measurement. I was listed on two of their publications [19, 81] that is based on the part-time work and collaboration with their team.

I would also like to thank the Ruxcon computer security team for their wonderful friendships, letting me work as a staff member at Ruxcon 2012, as a presenter on GCC plugins in 2011, and as one of their monthly lecturers. The plugin talk derives from the knowledge I have gathered while working on this thesis.

I would also like to thank Linux Journal and Linux Weekly News for publishing articles [14, 16, 15] that I have written while undergoing this PhD. The latter two publications derive from the knowledge I have gained while working on this thesis. I also thank the Linux Users of Victoria for
allowing me speak about writing and debugging Linux Kernel Drivers, and Linux.

Lastly, I would like to thank the Søndergaards, they are much more than friends, they have been like an adopted family to me. I am forever indebted to them.
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>A.7</td>
<td>Memory Management</td>
<td>214</td>
</tr>
<tr>
<td>A.7.1</td>
<td>New and Make Allocators</td>
<td>215</td>
</tr>
<tr>
<td>A.8</td>
<td>Control Flow</td>
<td>216</td>
</tr>
<tr>
<td>A.8.1</td>
<td>If-Then-Else</td>
<td>216</td>
</tr>
<tr>
<td>A.8.2</td>
<td>Switch</td>
<td>216</td>
</tr>
<tr>
<td>A.8.3</td>
<td>Loops</td>
<td>217</td>
</tr>
<tr>
<td>A.9</td>
<td>Higher-Order Functions</td>
<td>218</td>
</tr>
<tr>
<td>A.10</td>
<td>Defer</td>
<td>219</td>
</tr>
<tr>
<td>A.11</td>
<td>Concurrency</td>
<td>220</td>
</tr>
<tr>
<td>A.12</td>
<td>Compilers</td>
<td>223</td>
</tr>
<tr>
<td>A.12.1</td>
<td>Google</td>
<td>224</td>
</tr>
<tr>
<td>A.12.2</td>
<td>GCC</td>
<td>224</td>
</tr>
</tbody>
</table>

**Bibliography** 227
# List of Figures

2.1 Three level cache ............................................. 12  
2.2 Linux process in memory ..................................... 16  
2.3 Memory occupancy of a hypothetical process with ideal program analysis. ............................... 35  
2.4 Memory occupancy of a hypothetical process with imprecise program analysis. ............................... 35  
2.5 Memory unreachability: static vs dynamic reasoning ............................... 41  
2.6 Memory use: static vs dynamic reasoning ............................... 42  
3.1 General Go/GIMPLE syntax ..................................... 50  
3.2 Region constraint generation ............................... 58  
3.3 Creating a linked list in Go ..................................... 61  
3.4 Creating a linked list in Go with region annotations ............................... 62  
3.5 Runtime merging due to higher-order functions ............................... 78  
3.6 Runtime merging case ............................... 80  
4.1 Distinguishing object sharing ............................... 93  
4.2 “Memory-facing” fragment of the language ............................... 95  
4.3 The semantics of the memory-facing statements ............................... 98  
5.1 Original constraint rules ............................... 117  
5.2 Modified constraint rules ............................... 118  
5.3 Added constraint rule for closures ............................... 118  
5.4 Region data structure ............................... 120  
5.5 Example: copying two items to to-space ............................... 137  
5.6 Pointer into the middle of an array ............................... 139  
5.7 Duplicating data after collection ............................... 140  
5.8 Copying only the previously reachable parts of arrays ............................... 143  
6.1 Added rules for handling Go’s concurrency primitives. ............................... 165
6.2 Go-routine transformation ......................... 170
6.3 Producer/consumer example ...................... 173
6.4 Producer/consumer example with region annotations .... 174
Science is the belief in the ignorance of experts.

Richard Feynman

This thesis investigates the implementation and performance of an automatic form of memory management, region based memory management (RBMM), for programming languages. To better understand this concept, this thesis provides a design and implementation of an RBMM system for the Go programming language, where regions are statically inferred based on an object points-to relationship. Our design does not rely on a stack of regions to ensure memory reclaim safety, rather we introduce the concept of a protection counter for this purpose. Our analysis is of a flow-free and context-free design, which can reduce the amount of rework necessary when recompiling a source file. Go provides an ideal platform for studying RBMM since it uses garbage collection (GC) to manage memory. This interaction between our RBMM and a garbage collector provide a unique area of study. We later investigate an augmented RBMM design that introduces a region-aware garbage collector capable of reclaiming memory from a region.

Developing software is a challenging task. Sloppy coding, the stress of fast development (deadlines), language choice, and dealing with legacy code are all challenging aspects of the software development process. Further complicating matters is simply the fact that programmers are human, and thus are flawed and can easily make mistakes when writing programs. Such is especially the case when the language they are coding in requires the programmer to manage additional resources that are not directly related to
the problem at hand. One resource that programmers often find themselves having to manage is the use of the system’s memory. This task often requires the programmer to keep track of memory allocations, data types, and memory reclamation. Keeping track of such information presents an additional burden to the programmer, and increases the probability of bugs in the resulting program.

Programs can use a variety of data structures during program execution. When writing a program, the programmer cannot always determine how many instances of a certain data structure will be needed. For instance, a program might require additional data structures based on user input. Through the use of dynamic memory, the programmer can tell the runtime system that a new data structure instance is needed. The runtime system is responsible for fulfilling this memory request. This memory is where the object will reside. Certain languages also insist that the programmer reclaim memory for those objects which are no longer needed. This creation and reclaim of memory is known as memory management. If the memory is not reclaimed, the system can run-out and the program will prematurely terminate. Similarly, reusing memory that has been reclaimed can also problematic, since that memory might be used by another aspect of the program or runtime system.

If memory is improperly managed, the software can create a variety of problems. Insufficient use of memory can create leaks, and put a performance strain on the system. Similarly, invalid use of memory can lead to the software crashing or producing invalid output, compromising the integrity of the software solution. While it is annoying to have a program such as a text editor or web browser crash, in mission critical environments (e.g., healthcare or aviation) a software failure can have serious consequences. For instance, on January 22, 2004 NASA’s Mars rover, Spirit, suddenly became non-responsive to its operators’ commands [82]. This was the result of a memory limitation. The system responsible for manipulating the rover ran out of storage (128MB of RAM), leaving Spirit not properly responding to
NASA’s commands. Spirit’s mission manager, Mark Adler, later wrote on his blog that NASA corrected the problem by “effectively reformatting” the rover’s flash memory [2]. If improperly managed memory can affect NASA, surely it can have an impact within other mission critical contexts.

This thesis explores region-based memory management (RBMM), which is an automated form of memory management, that relieves the programmer from the complicated aspects of manual memory management. We study this memory management strategy from the context of the Go programming language. Go makes for an interesting platform to study from since it is garbage collected by default, and our goal is to study an alternative memory strategy, RBMM. This thesis tries to reduce the amount of work that the runtime heavy garbage collected version of the language induces by introducing an RBMM system into the language’s compiler and runtime system.

RBMM is not as common a feature in programming languages as garbage collection (GC), which also relieves the programmer of having to manage memory, but at the cost of a higher runtime performance over that of manual memory management [45]. The trade off between these two solutions is runtime performance versus efficient use of memory (keeping as little memory resident as possible). GC performs its work at runtime by periodically scanning memory and reclaiming any allocated data that is no longer needed during execution. In contrast, RBMM is the result of a compile time analysis, whereby the compiler inserts memory reclamation operations into the program. RBMM offers the promise of easing the programmer’s burden while also offloading the memory management analysis to the compiler. It has the potential to reduce the need for an otherwise expensive runtime task.

At best, both of these solutions approximate the state of the program. GC conservatively approximates runtime state by scanning the memory it has allocated for unreachable items. RBMM conservatively approximates what memory can be reclaimed at each program point. The latter is of interest to this thesis, which explores not just an RBMM solution, but a
solution that also incorporates the use of a garbage collector to manage items whose lifetime cannot be safely approximated at compile time.

1.1 The Programmer’s Burden

How a programmer manages memory depends on the programming language being used. Many languages (such as C and C++) require the programmer to request memory and also reclaim that memory at a later time. But humans are notoriously bad at this, and the result is unstable programs that either crash or exhaust the system’s memory resources via memory leaks.

One way memory considerations complicate a programmer’s job is that it can be difficult to establish the lifetime of an allocated memory block. Often memory is requested in one function for use by another function. Moreover, such an allocation can be reclaimed in a completely different function. This disjunction between allocation and reclamation sites means that the programmer must employ *global* reasoning when thinking about memory. This idea contrasts with *local* reasoning, whereby the programmer can assume that memory is not needed outside of the function it has been allocated within. Global reasoning can be more taxing for the programmer, which can result in error-prone programs. The programmer must be fully aware of all functions his or her program calls (directly or indirectly), and if they require any memory associated with returned values to be reclaimed. This means that a function not written by the programmer must explicitly state, in documentation, the need for any memory reclaim that a function might allocate. Global reasoning is required when the programmer makes explicit requests for memory, also known as *dynamically allocated memory*, and then passes that memory to other functions. Dynamic memory can be returned by the requesting function and passed as data to other functions. In other words, dynamic memory can escape the function it was created within. Therefore, dynamic memory can be accessed from any point in the
program; it has global context. This idea contrasts with an easier form of memory management known as *local memory*. This form of memory is often implemented as a stack in the process’ memory space, which grows and shrinks during execution. Such memory requires the programmer to apply simple local reasoning. Stack memory should never escape the function it is allocated within (it is local), and is almost always managed implicitly by the compiler when the program is being compiled. Since stack memory is only valid for the lifetime of the function (allocated at function prologue and reclaimed at function epilogue), any use of it after function exit can result in the program processing invalid data and producing bad results or crashing.

In contrast, since dynamic memory is not always localized, programmers must determine when data associated with a reference \( r \) is valid and when it should be reclaimed. Once memory is reclaimed, the programmer must also ensure that it will never be accessed again via \( r \), as that memory might subsequently have been reused for a different purpose.

Automatic memory management aims to reduce the programmer’s burden and increase software reliability. An automated memory system will try to reclaim allocated memory without the intervention of the programmer. For instance, the Java programming language does not require the programmer to release memory. Instead, the language and its accompanying runtime environment use GC to detect when an object is no longer needed. GC is a runtime operation which works by scanning the program’s memory to identify allocated objects that can be reached from the live objects (the roots). Any allocated object that can be reached from a root must not be reclaimed, as this would remove an object that might be used later during the program’s execution. However, unreachable objects will have no role to play in the rest of the program’s execution and therefore can have their memory reclaimed by the garbage collector. This runtime analysis is not always an ideal solution, as it can increase a program’s execution time.

Due to the additional cost of running a garbage collector, some people argue that this often non-deterministic task of memory management via
GC is not ideal for system level programming, including real-time systems. System level languages (e.g., C, C++) offer much programming power to the developer, at the cost of reducing program safety (e.g., they present the programmer with resource management tasks). These languages are commonly used for operating system kernel development, driver writing, and other low-level computation tasks. Arguably, these are also the same uses that need the highest level of safety, including memory safety.

1.2 Our Goals and Solution

Surely we can do better than this. A pertinent question then is: Can we have a system level language that is both memory safe and efficient? In 2009, Google introduced the Go programming language that aimed at being a memory safe system level programming language. Similar to Java, Go provides a garbage collected runtime environment. Thus, a large portion of the memory safety of Go arises from the fact that the language relieves the programmer of the burden of making decisions on the lifetime of memory allocations, which is handled by the collector. However, this comes at the price of a runtime-based solution for memory management. But there is an alternative to GC, called RBMM. Instead of waiting until runtime to locate unreachable allocations for reclaim, an RBMM system works by performing a static analysis of the program's source code and inserts memory management operations during compilation. This approach avoids the overhead generated by runtime memory scans that garbage collectors require.

This thesis investigates RBMM as a solution for efficiently managing a program's memory while eliminating the faults induced by manual memory management. We strive to answer the following questions from the context of the Go programming language environment:

- How can we transform a Go program into one that supports RBMM and also not require any additional programmer annotations?
- What analyses do we need to implement RBMM?
• What runtime operations do we need to implement to support RBMM?

• Can we create an automatic memory management system that avoids the negatives of RBMM and GC while attaining the benefits of both systems?

• What is required to implement an RBMM system for a concurrent language?

This thesis contributes an investigation of incorporating a fully-automatic RBMM system to the existing Go language, that is also capable of operating within a concurrent environment. For the reader not familiar with such topics, language and memory system background information are provided in Appendix A and Chapter 2 respectively. Our RBMM design for a first-order subset of Go is presented in Chapter 3. Chapter 4 provides a formal semantics to reason about the transformations that our RBMM design requires. Chapter 5 introduces our design for a combined RBMM and GC system, which does not have to perform a complete scan of all of the variables in the program. Chapter 6 investigates incorporating RBMM within a concurrent context. Chapter 7 provides a literature review of related work. Chapter 8 concludes with a discussion of future improvements.
Chapter 2

Memory Management

A clear conscience is the sure sign of a bad memory.

Mark Twain

Managing a system’s use of memory is a common task in software engineering. Having a basic understanding of the various types of memory in a system is useful for understanding how a program can influence the system’s performance via this managed resource. Since memory is a limited system resource, properly managing it is important for developing safe and efficient programs. System crashes can occur if memory is not managed correctly. Improperly managing this resource can reduce the amount of memory available for other processes to use. This chapter introduces some basic concepts of memory management, as well as region-based memory management (RBMM).

2.1 Introduction

Memory is a limited resource on computing platforms. Even on modern systems with gigabytes of memory, poor performance can result from abusing these resources. While systems are increasing in computing power and gaining larger main memory stores (increasing random-access memory (RAM) size), more processes are being executed on these machines in parallel. With more processes being run and consuming additional memory, applications need to be written in a way that will not “hog” all of the system’s memory. Inefficient use of memory can have a negative performance impact on other
simultaneously executing applications. In other words, the fact that a ma-
chine has more memory does not permit programmers to manage memory
as if it was an unlimited resource.

Application portability is another factor for programmers to consider
when writing their programs. Having a single application that executes
in multiple environments is very useful, since it only requires sharing an
existing binary or recompiling the code to run on a different architecture.
While one machine might have vast quantities of memory, a smaller, possibly
embedded device, can have tighter memory constraints. Therefore, sensible
use of resources is necessary so that both systems can execute the program
without being severely constrained on memory.

2.2 Background

2.2.1 System Memory

A computer is composed of multiple types of memory: disk, system, cache,
and registers. This section introduces these various types of memory stores.

External Memory

External memory is the slowest data store within a system. Traditionally
this store was called “disk” memory; however, solid-state (no spinning disks)
drives are becoming common on modern systems. Typically, data stored
on an external drive are non-volatile. This means that the information is
preserved until the user asks the operating system to remove it (or until a
malfunction occurs). External memory is located furthest away from the
CPU, such as on a hard drive or an externally connected store (e.g., USB
drive). Not only is this memory the furthest from the CPU, but it is the
slowest for accessing a particular piece of information. Given a hard disk,
the drive must first move its read/write head to a specific location and then
transfer that information across a data bus to the CPU. The amount of
time it takes for the drive to locate the data is called *seek time*. A benefit of an external drive is that it is non-volatile, can be the lowest in price, and most abundant in capacity. Binaries, text files, and media all reside on these stores.

**System Memory**

When the operating system (OS) loads a program to execute, it must first locate the binary on the non-volatile external store. That program is then copied into system (also known as “main”) memory. This memory is the RAM, which is volatile, solid-state, and is faster and closer to the CPU than external memory. Volatility means that the information stored is only temporary. Upon system reboot or power-down, the state of the memory is erased, and all of the information is lost. Solid-state is an important trait. Unlike a traditional hard drive, there is virtually no seek time since there is no read/write head that has to move in order to locate data. In contrast to external memory, RAM is more expensive and has less capacity. When a system becomes low on system memory, data can be exchanged with disk to make room for the currently executing program(s). This process is known as *paging* or *swapping*. Exchanging data between the solid-state RAM and disk (which can also be solid-state) is a slow process. Therefore, having memory efficient programs is necessary for optimizing system performance. It is important to mention that while solid-state hard drives are becoming more common, they are still slower to access than RAM, due to data having to traverse a longer bus path to reach the CPU.

**Caches**

The next tier of memory is the cache. This memory is used to store recently accessed data, so that future accesses will be faster than having to go to RAM or disk. The cache is solid-state and is located on the CPU die, therefore it is faster for the CPU to access than the RAM. In contrast to the system memory, the cache is more costly and of smaller size.
This thesis is not concerned about the details of caches; however, data locality influences system performance via cache behavior. Locality refers to how closely a piece of data is stored in relation to other data. A system will operate more efficiently if the data it is looking for lies within the system’s cache. If the data is not located in the cache, then the RAM, and possibly disk, must be searched. This can be costly. The results of our memory management research does impact cache and overall system performance, therefore having a basic understanding of caching is useful when trying to make sense of our research.

Caches are typically implemented in multiple levels. In this section we will consider a three-level cache, which is common on the modern Nephalem Intel CPUs; however, other architectures implement caches differently. Figure 2.1 illustrates this design\(^1\).

The lowest level of cache, level 3, holds both data and instructions and is shared between CPU cores given a multi-core system. The level 2 cache is not shared between CPU cores and also contains both data and instructions. Even closer to the CPU is the level 1 cache, which is divided into instruction and data. The most recently accessed information is stored here. When the level 1 cache gets filled, the older/least-recently-used data is evicted and stored into a lower level cache.

When a program requests data and that data already exists in cache, a

\(^1\)The concept for this graphic was based on the image displayed in [1].
hit occurs. The system is most efficient when the cache is hot, whereby a majority of data requests can be fulfilled by the CPU quickly obtaining the requested information from cache. If the data is not in cache, a miss occurs. The CPU must then access the RAM for the data. If the data cannot be found there, it must be obtained from the external store.

**Registers**

When needed in computations, data must reside directly in the CPU registers. This temporary store is the fastest of the types discussed above, but is also the most limited in capacity. Since modern CPUs are capable of executing multiple programs in parallel, the register state, which pertains to a specific execution, must be swapped out to the cache allowing the CPU to restore the state of another concurrently executing program, to perform computations on that program’s behalf.

### 2.2.2 Virtual Memory

Virtual memory is an abstraction of the available memory that a process can have. The OS presents each process with a seemingly contiguous chunk of memory. In fact, it is often the case that the OS will present the process with more memory than the system memory actually contains. From the perspective of the process, the memory is contiguous. However, the reality is that the OS might have presented the process with chunks from non-contiguous address spaces, but the process does not know any better. Virtual memory works because of virtual addressing. All processes execute within the same contiguous virtual address range. The address range that is seen by each process is not unique, and is the same for all processes. The OS is responsible for mapping the process’s virtual addresses to physical/logical system memory addresses. The OS is also responsible for the safety/partitioning of memory, to prevent processes from accessing physical addresses that belong to other processes. Virtual memory will be discussed in more detail in Section 2.2.3.
Paging

As previously mentioned, a process operates within a virtual address space. What happens if a process tries to access an address or needs more memory than the system can provide? In this case, the OS is still responsible for giving the process more memory. Since the external memory can be used as a giant temporary store, the OS can obtain additional memory from a swap file or partition located there. Data on the external store can be swapped with system memory to provide additional system capacity and allow the process to continue its execution. A page fault occurs when a page of memory is not located in system memory and the OS has to obtain the data from the external store. Page faults are expensive, since retrieval and copying to/from external memory is slow (especially if the external store is a disk drive).

2.2.3 Process Memory Layout

A program exists as an executable file on a computer’s external store. Before program execution can begin, that file must be copied from the store into the system memory. It is the responsibility of the OS’s program loader to map an executable file from the external store into the system memory. Before we look at how memory is requested by the programmer, it is useful to envision how a program looks at runtime after it has been loaded into memory.

Process Virtual Memory

On modern machines, each program is given its own portion of system memory to execute within. This memory contains the program’s instructions, variables, as well as a segment of memory where dynamic allocations can be produced from. The latter segment is known as the heap. On a Linux system, all processes operate within a virtual address space. This range is identical for all processes, so that each concurrently running process will appear to utilize the same address range. This range is architecture specific,
but for x86 CPUs running Linux within a 32-bit address space, each process is given a virtual address space of $2^{32} - 1$ bytes (or 4GiB). These addresses are virtual. Even though the addresses might be the same for each process, the underlying OS memory addresses are completely different and do not overlap. The OS is responsible for translating between a process’s virtual address and a physical address in the system. Virtual addressing allows all processes to operate on a seemingly contiguous (linear) span of memory, even though that might not actually be the case. The physical system memory is divided into pages. When a process needs additional memory, any available page is mapped into the needy process. While the physical memory might be out-of-order, or even fragmented, the process which operates on virtual addresses does not witness this fragmentation. In fact, the system might not even have the total amount of memory that is represented by the virtual address space. As more system memory becomes available, which can occur when processes terminate, their memory can be reused for other processes. External memory can be used for paging, which provides additional storage when the system memory becomes low. When paging occurs, portions of the main memory are saved/copied to the external store for later use. The addresses of the “paged” data in system memory can be reused to store new data. Paging from system to external memories can be slow, since the data has to be copied to a slower store located further from the CPU.

Figure 2.2 illustrates what a process looks like in main memory on a Linux system. It shows the various memory segments that comprise the virtual address space of a single process running in a Linux environment. All concurrently running processes look similar, and their address space has the same range of virtual addresses; however, the sizes of the individual segments might vary. The following describes the memory segments of a process, beginning at the text segment and working towards the higher addresses.

**text segment** This segment contains read-only object code which represents the instructions for the program to execute.
**data segment** This segment contains the globally declared and defined variables.

**bss segment** The block started by symbol (bss) segment contains all static global and local variables declared in the program. Their initial values are all zeroed at the time program execution begins.

**heap segment** This segment grows toward the stack (towards the higher memory addresses) and is where dynamically allocated data are kept. Typically a pointer to this data is the result when the programmer asks the operating system for memory at runtime (**e.g.**, malloc).

**memory mapping segment** This segment contains memory that can be used for creating a custom memory allocator or can be used to map
files from disk into system memory. The latter use reduces disk reads when accessing data from disk, since that portion of the disk is copied into system memory.

**stack** On an x86 architecture the stack grows downwards. That is, as items are pushed onto the top of the stack, the stack will extend towards the lower memory addresses of the process, approaching the heap. Data for local variables and formal parameters are stored here. Data stored on the stack are temporary and last for the duration of the function call that is being executed.

**kernel space** This is a protected memory segment used by the OS to map in code and data for use only by the kernel. For the 32-bit x86 Linux kernel, this space actually contains a copy of the entire kernel, which improves cache performance [83].

### 2.2.4 Stack Data and Stack Frames

Many portions of a program utilize an amount of memory that can be determined statically at compile-time. Variable declarations are a programmer’s way of expressing to the compiler that the program needs a specific amount of memory. Variables declared within a function are often termed automatic, or local, meaning that they are only accessible when that particular function is being executed. The formal parameters for a function also are included in this set of variables. When a compiler processes a function, it can calculate the number of variables and their sizes that make up that function. Therefore, the compiler knows how much memory will be needed by the system to execute any function. The memory for these variables will be produced from the stack segment of the process.

When a program begins execution, memory needed for local variables in each function must be obtained from somewhere. When a function is called, the stack segment associated with the process is expanded to hold enough memory for the locally allocated variables and formal parameters
of the function. This chunk of space is known as the activation record, or stack frame, for the call. Hence, the stack segment of a process consists of a sequence of different sized frames.

When a function calls another function, the stack frame for the callee will be added on top of the stack. As functions are called, the stack will grow. The stack frame for the currently executing function will always be at the top of the process’s call stack. When a function completes, the stack will shrink, effectively popping the frame for that function (its local and formal parameters) from the stack. The execution will then resume on the lower frame (the caller). This lower frame becomes the new top-most frame.

2.2.5 Dynamic Memory

When writing a program, it is common for the programmer not to know how much memory will be needed to accomplish a specific task. In these cases, memory can be requested from the OS via an allocation call, such as new or make in Go, or malloc in C. The memory returned from such allocation calls is produced from the heap segment\(^2\) in the process’s memory space. Memory allocated from this segment is not reclaimed once the function terminates, and can live across multiple function calls and returns. The memory may live throughout the lifetime of the program, although this is often the sign of a memory leak (memory that is allocated but never reclaimed, even after its last use).

Suppose a program takes an arbitrary integer value as input, and that the program makes use of this value by creating a list structure of data based on the value supplied. In this case a programmer does not know the amount of memory needed to construct the list. Instead, the programmer must write code to create the list dynamically at runtime. Consider the following piece of Go code:

\(^2\)Depending on how a memory allocator is designed, the memory might come from the memory mapped segment instead of the heap. We refer to both the heap and memory mapped segments interchangeably as segments which can be used to produce dynamically allocated memory.
The routine above creates a linked list of an arbitrary length based on some input value `num`. While the use of a `slice` in Go would be more appropriate here, the example above illustrates a common way in other languages to create a linked-list structure of an unknown length. In languages that do not manage the reclamation of memory automatically, such as C, the memory for each allocation call would have to be explicitly reclaimed once the list is no longer needed. If the memory were never reclaimed it would cause a leak and the program could later run out of memory. In this case Go’s garbage collector is capable of reclaiming this memory after the function returns, if `head` is not reachable from any variable in the program at the time a collection takes place. If a reachable variable, even if it is unused, points to `head` then the collector cannot reclaim the list’s memory.

When analyzing the lifetime of objects associated with an allocation, it is important to consider that such allocations can occur from separate program libraries or modules (object files). To obtain a fully accurate picture of the lifetime of an allocated object, a whole program analysis must be performed.
Often, the source code for a particular library is not available, therefore a compiler cannot perform a complete lifetime analysis. What this means is that a programmer who is responsible for memory reclamation must also be aware of any allocated memory that results from calling external libraries. In such a case, a compiler cannot produce warnings as it does not have a complete representation of the program. A complete representation would require some metadata or source code for any of the external functions being called. Dealing with external libraries complicates the programmer’s job of writing resource-safe applications.

2.2.6 Dynamic Allocation Problems

The manual management of memory is a complicated task for programmers. The allocated memory can be reachable across multiple function calls, program modules, and libraries. Thus the programmer must be aware of when the memory was allocated, where it was allocated from, and when it can safely be reclaimed. Even if the language automatically reclaims memory, the programmer should still be aware of other problems that dynamic memory allocation can cause, such as leaking memory, dangling pointers, and referencing invalid or out of bounds memory.

Fragmentation and Locality

A memory allocator must return a contiguous chunk of memory to the process. This is because the compiler, when generating code, will assume that each individual object or array are placed in a contiguous memory space. If not, the data for a single object (or array) would be distributed throughout the process and access to an element or member of an object would not produce the proper value. Consider the case of a 32-bit integer that is split between two disjoint (non-contiguous) memory blocks. A reference to the integer would only produce the first half of the integer and not the second half.
During program execution, subsequent allocation and deallocation of memory can produce “gaps” within the memory space of the process. These gaps represent freely-available blocks of data that can be recycled by the memory allocator and given back to the program to fulfill later allocation requests. Fragmentation is a property of the underlying allocation routine (e.g., malloc) and can result from the process returning unused memory in a different order from when it was allocated. Similarly, a fragmented memory space can arise from the program requesting a lot of memory. For instance, a highly fragmented process will have non-contiguous gaps of freed memory within its memory space. If a subsequent allocation is issued and none of those free-gaps are of adequate size (and contiguous) to fulfill the request, then the allocator must request more memory from the OS.

A contiguous memory space is ideal for system performance. For instance, an optimal memory layout would have complex objects (a structure and all of its fields) located within close memory proximity. Optimal locality reduces the likelihood of the system to experience cache misses, or even page faults, when the program references an address. Optimal locality can also mean that a process is using its memory space efficiently. The latter would avoid the need for the OS to obtain additional memory to fulfill an allocation request for a contiguous block of memory, even if the sum of the (non-contiguous) free-blocks could otherwise have met the request.

Memory Leaks

Memory leaks occur when a process requests memory and never returns it to the system, even after the memory is no longer needed. These drain a system of its available resources, which can impact the performance of other concurrently executing programs. Leaks commonly occur in languages which require the programmer to manage the reclamation of allocated memory manually. If the memory is never reclaimed, then it will be unavailable for other uses.

Complicating matters is the fact that allocated memory can escape the
function it was allocated in. Variables that point to allocated memory can be passed as data to other functions, therefore it is not easy for the programmer to determine the last use of an allocated object for reclamation purposes.

System memory is a global resource used by all executing programs. Even though OSs can prevent the sharing of memory between processes, the memory itself is still a limited resource that the OS is responsible for distributing to all processes. Any excessive use of memory by the programmer can greatly impact the system as a whole. Preventing memory leaks is necessary for maintaining a stable system.

**Invalid Memory Access**

Invalid memory access occurs when the program attempts to make use of information stored at an address in memory that is out of bounds to the process, or if the program interprets data at an address incorrectly. These cases occur in languages that permit pointer variables, such as Go. These are well known bugs often resulting from poor programming by the human, such as attempting to access array data outside the bounds of the array. The following Go example illustrates this case.

```go
func foo() {
    var values [100]int
    println(values[1000])
}
```

There are two problems here. First, the address containing the value at index 1000 of the `values` array might be in a different memory segment or even outside of the memory allocated for the process. The second problem is that reading such data is incorrect. In this example, that value will be interpreted as an integer, although it could be anything. Further, that value does not belong to the `values` array.

The Go language attempts to reduce these invalid accesses by implementing bounds checking for arrays and slices. In this case, the compiler
will statically check if 1000 is larger than the defined array size of 100. The compiler will issue an error and not produce a faulty executable. Go also performs runtime bounds checks for dynamically sized arrays (i.e. slices), and if an access violation occurs, the program safely terminates.

Pointer arithmetic is also dangerous, since it makes creating invalid memory access all too easy for the programmer. Go does not permit pointer arithmetic, as adding/subtracting offsets to a base address can result in an invalid memory access.

**Dangling Pointers**

Dangling pointers occur when the pointer variable or value in a data structure contains the address of dynamically allocated data that has been reclaimed. In such a case the pointer might contain the address of another object (possibly of a different data type) or a bit-pattern that is not an address (e.g., a value contained in a primitive). If the pointer variable is never updated to reflect this change, it will point to memory that is no longer relevant. The OS will issue a segmentation fault when a variable containing an address located outside the process or within a protected memory segment is dereferenced. If the fault is not handled properly the program can prematurely terminate. A dangling pointer is only problematic if it is dereferenced.

Dangling pointers are possible in languages that require the programmer to manually manage memory and also in languages that do not guarantee initialized allocated memory. Languages that automatically manage memory reclamation (e.g., garbage collected and some region-based memory management languages) do not suffer from this problem; however, the language and its runtime system must be carefully designed to prevent these cases [26]. Since automated memory management systems ensure that all reachable variables can only reference other reachable data, they are immune from the problems caused by dangling pointers.

The following example, in the manually managed C language, illustrates
the access of a dangling pointer:

```c
void update(Node *n)
{
    /* Make use of n ... */
    free(n);
}

void make_node(void)
{
    Node *n = malloc(sizeof(Node));
    update(n);
    n->id = 42; /* Problem */
}
```

In this example the variable \( n \) is allocated enough memory to represent a single instance of a Node. \( n \) is not the object, but a variable that points to a Node instance in the heap. The `update` function uses \( n \) and then calls `free` to release the memory, referenced by the value located at \( n \), to the memory allocator. \( n \) then becomes a dangling pointer and any access to data via \( n \) is a memory violation.

While languages with automatic memory management are without the problems of dangling pointers, they are not devoid of the problems caused by \textit{nil} value references. Some languages (such as Go) have a \textit{nil} value which can be used to set/initialize pointer variables to a value of nothing (they point to the address 0). Since these pointers point to nothing, reading or writing to \textit{nil} is a memory violation. Setting a pointer to \textit{nil} is also a hint to the garbage collector that the data being pointed to is no longer accessible to the program. If nothing points to an allocated object, then the memory for that object is no longer needed and can be reused for later allocations.
The following Go example illustrates the access of a nil pointer:

```go
func setDefaultID(node *Node) {
    node.id = 42 /* Problem */
}

func main() {
    n := new(Node) // Create a pointer to a Node object
    n = nil
    setDefaultID(n)
}
```

The Go language does not have an explicit memory reclamation operation, as it is a garbage collected language; however, programmers can influence the results of GC by “zeroing” or “nulling” a pointer variable. In the example above, n is passed to a function that updates the Node instance. However, n was “nulled” and thus accessing the id field will be accessing an invalid address. When execution reaches the body of setDefaultNode, node is “nil” (0). This function will try to reference the id field, which is just an offset from the base of node. In this case, the access of the id field from an address of 0 will result in a segmentation fault.

### 2.3 Automatic Memory Management

The problems discussed above strengthen the argument that memory management can be a complicated task for programmers to accomplish. With that thought in mind, there are a few solutions that aim to remove the need to explicitly manage memory from the programmer’s role, and consequently reduce the probability of creating bugs. We now discuss solutions to the memory management problems presented earlier.
2.3.1 Region-Based Memory Management

Region-based memory management is a combination of static analysis and runtime instrumentation. The seminal research in automated RBMM (involving compiler-driven region inference) is typically associated with work done in the 1990s by Mads Tofte and Jean-Pierre Talpin [78, 79]. Through a process of region inference, a compiler can determine how long objects live and which objects have similar lifetimes. The compiler can then group related objects together such that all of their memory is allocated from the same (often contiguous) chunk of memory. This group of objects is said to be allocated from the same region. Since these objects live together, they can also die together. Based on a “region inference” static analysis, the compiler can transform the program by inserting function calls to region operations. These operations are responsible for managing the creation and removal of regions, and allocation of data from regions. Region operations form the basis of an RBMM runtime system.

The following is a list of region operations that a typical RBMM system implements in its accompanying runtime system.

\textit{CreateRegion()} Create an empty region from which data structures can be allocated.

\textit{AllocateFromRegion}(r, n) Allocate n bytes from region r.

\textit{RemoveRegion}(r) Reclaim all of the memory from region r so that it can be reused later.

RBMM Benefits

One benefit of RBMM is that it can significantly reduce the execution time needed to reclaim the memory associated with objects. The region containing the memory for all of the related objects can be reclaimed all at once. This can be much faster than visiting each object and reclaiming its memory individually.
Another benefit of RBMM is that it can enhance cache locality. Since a relation is established at compile-time, which determines what objects belong to which regions, an access to any one of the region’s objects can indirectly cause the cache to page-in the other objects from that region. This means that the related objects, which also might be accessed at a similar time, are likely already to be in the fast system cache and not have to be retrieved from RAM or disk.

RBMM Limitations

While all of the benefits sound like a great choice for languages to utilize, RBMM is not without its drawbacks. Since a region is reclaimed all at once, all of the objects in that region must no longer be needed (e.g., no other reachable objects reference the objects in the region that is to be reclaimed). This constraint can create the situation where a majority of a region’s objects are no longer needed, but a few (or even one) object remains reachable. The RBMM runtime system cannot remove this region until all of its objects are no longer used. This is what we call region bloat. An ideal RBMM system will avoid this situation as best it can, to lower the memory pressure of the system.

RBMM is based on a static analysis, therefore there are times where this analysis cannot distinguish the lifetime for all items. This limitation can result in most of the memory allocated by the program being allocated from a single giant region, which cannot be released until the end of the program’s execution.

In such cases RBMM does not reduce the program’s memory requirement at all. Figure 2.4 illustrates this kind of memory leak. Each kind of behavior is observed in practice [69]. RBMM systems can also yield larger binaries, due to the inclusion of primitive region operations. Besides the time needed by these calls, the increased size can affect instruction cache performance.

Lifetime analysis poses another challenge for RBMM. For instance, if a dynamically created item is reachable from a global variable, the compiler
cannot determine the lifetime of that item, and must therefore conservatively assume that its lifetime is the lifetime of the program. Global variables can therefore create memory leaks.

Another problem is that in the quest to avoid having too many small regions, the RBMM system may put into the same region items that in fact have different lifetimes. The problem is that while some items in a region are reachable, no part of the memory of that region may be reclaimed. This means that an RBMM system cannot reclaim the memory of a dead item in a live region, which results in region bloat as discussed above. This problem does not exist in GC systems.

**Automatic and Manual RBMM**

RBMM can be implemented as either a manual or fully-automatic system. In the former, the programmer is responsible for inserting the region operations, which reduces the need for a static analysis. In addition, manual systems require the programmer to determine which objects are allocated from which regions. Manual systems are similar to traditional manual memory management. Such systems add additional complexity for a programmer, since the programmer must be aware of when groups of objects are no longer needed and when no other objects refer to objects in a reclaimed region. In automatic RBMM systems, which is the primary focus of this thesis, the compiler determines all of the object relationships and safely approximated lifetimes, and transforms the program accordingly. Berger, Zorn, and McKinley [7] provide a thorough investigation into the world of manual memory management, including regions, in their 2002 research.

While a fully-automated RBMM system requires no programmer intervention, it may favor certain programming styles. A programmer's style can influence the compiler to make certain decisions about how regions are managed [79]. For instance, a programmer can write a program such that global variables point to other variables within the program. Global variables can complicate static analysis. For instance, global variables have a
lifetime that lasts the duration of program execution. Therefore, objects that are pointed to by a global variable must also live for the lifetime of the program, or until that variable no longer points to them. Any regions that those objects belong to must also remain alive, causing region bloat. Yet a fully-automated system saves the programmer from having to remember when to reclaim objects.

**Region Allocation Strategies**

When designing an RBMM system, a key decision that has to be made is how the runtime system manages the set of all regions. Certain RBMM systems are implemented in a way that permit pointers between regions. While this can reduce the overall size of a region, and potentially lead to faster region reclamation, care must be taken to prevent dangling pointers [26]. For instance, if a region is reclaimed and live data references objects from that region, then those references will become dangling pointers and refer to invalid memory. A region cannot be reclaimed until all of its objects are no longer pointed to by objects from external regions. This constraint imposes a lifetime on the regions. One solution to such extra-region pointers is to allocate regions as a stack [78, 63, 12].

In the stack-of-regions approach, regions are created and pushed onto a stack. The top-most region on this stack is the youngest, and the oldest is at the bottom. This system can impose a one-way direction of pointers; pointers can only point to objects within the same or older regions [58]. In such a system, the regions are reclaimed as a stack pop operation. This ensures that a region will never contain dangling pointers [26].

Another approach to region management is to nest regions in a tree hierarchy. Regions at a higher level, the parents, cannot be reclaimed until their child regions have been reclaimed. This solution has been utilized for RBMM systems that function in a concurrent environment, whereby the parent node holds a concurrency-lock on its children [30]. Access to a child region can only occur if the parent has unlocked the child.
The solution we present in Chapter 3 is based on the data types of allocations. However, our analysis is different from Tofte and Talpin in that our region inference is based on a points-to relationship between objects. Their system is based on a type and effect system of expressions which determines region inference. Their design results in a stack of regions, whereas our solution generates regions that contain all of the objects that can access each other.

2.3.2 Garbage Collection

A common system of automatic memory management is garbage collection (GC) [51]. GC is primarily a runtime operation which traces all reachable pointers and reclaims the memory for objects that cannot be reached. These systems are more runtime expensive than RBMM systems since their analysis is performed during program execution. In contrast to RBMM, GC systems do not suffer the problems of objects that do not have statically-decided lifetimes (e.g., global variables). The concept of GC can be attributed to John McCarthy who, in the 1950s, discussed its use within the context of the LISP programming language [60].

Root Set

A garbage collector first begins its operation by scanning a set of addresses which are reachable in memory. The starting point, the root set, consists of stack variables, registers, and global variables. At any time during program execution, these data are accessible. When a collection cycle begins, the garbage collector will transitively follow non-nil valued pointers starting from the roots until a nil value address is reached. This means that any objects not visited by the collector cannot be reached by any live variables in the program. Therefore, the memory associated with those non-reachable objects can be reclaimed.
Non-Moving and Moving Collectors

There are two primary approaches to reclaiming items that are determined to be garbage. A *non-moving* collector passes over all garbage items, linking them together in a freelist. Future allocations are then taken from this list. In contrast, a *moving* collector consolidates all the non-garbage items, typically into a small number of contiguous regions; the remaining memory is then free to be allocated later.

Moving collectors have several advantages. First, they allow memory to be quickly allocated by simply advancing a pointer. Second, they give greater locality of reference. Third, they naturally defragment free memory as they consolidate in-use items; non-moving collectors must explicitly do this as a separate operation. Finally, the time taken to consolidate non-garbage items is proportional to the amount of non-garbage, while the time to link together the garbage items is proportional to the amount of garbage. In a well-designed GC system, the amount of non-garbage will *usually* be small compared to the amount of garbage.

A copying garbage collector can be viewed as a moving collector. Copying collectors divide the memory of the process into spaces called *semispaces* [27]. We consider two spaces for the discussion here, but other collectors, such as *generational*, can divide the memory further. These semispaces are called the *to* and *from* spaces [5]. As a program executes, all of the allocations produce memory from the *from* space. During a GC the allocated items reachable from a root node are copied from the *from* space into the *to* space. Once the collection is complete the *to* space becomes the new *from* space. Any item not copied during the collection cycle must have been unreachable from a root node, and therefore the item’s memory will be recycled for future allocations.

Conservative and Type-Accurate Collectors

As it scans memory items, the GC system must determine which values are pointers to memory items and which are something else (such as primitive
values). A conservative collector makes this decision by looking at the value. Since a bit pattern that represents an address in a part of the heap managed by the GC system could be a pointer, conservative collectors treat it as a pointer, and keep alive the item it points to, even if that item is an integer or other primitive type. A type-accurate collector maintains type information about every variable and every structure type, and uses this to decide which values are pointers.

Conservative collectors are generally simpler, since they do not need to consider types. Therefore, they do not need the cooperation of a compiler to provide type information. They are also applicable to weakly typed languages, such as C, in which (due to typecasts) values present at runtime may not reflect the declared types of the variables holding those values. However, conservative collectors can mistake integers or other non-pointer values for pointers. Such mistakes can accidentally preserve an item, and all the other items reachable from it, which may collectively represent a large amount of memory. Since these collectors cannot be certain whether a value (bit pattern) they treat as a pointer actually is a pointer, they cannot update the value, which means that they cannot be moving collectors.

Stop-the-World and Concurrent Collectors

One issue with GC is that a collector must not mutate values that are concurrently being read from or written to by the mutator (the non-garbage-collector portion of the program). This same issue complicates concurrent programming. The simplest solution is for the collector to pause program execution (stop-the-world), including all of the process’ threads that might be concurrently executing, and then perform the actual collection. This method can considerably slow down program execution, as all mutator threads must wait until the collector has finished before they can resume execution.

Parallel and concurrent collectors are designed to avoid halting the mutator for significant amounts of time. Parallel collectors work by scanning the mutator’s memory in multiple threads, effectively distributing the pro-
cessing of the GC task. In contrast, concurrent collectors perform their work while the mutator is simultaneously executing. If the GC can guarantee that certain data will never be written to (and possibly read) by the mutator during a collection cycle, then it can safely process that data. Concurrent collectors which do not modify the data of the objects (non-moving collectors) can permit reading of the objects from the mutator [51]. However, write access to an object must be locked to prevent the mutator from overwriting data by the garbage collector or vice versa. In the case of a multi-threaded program, where a thread only has access to data for itself (e.g., thread local storage) and is not being written to by another thread, then the memory associated with that single thread can be collected while the rest of the program concurrently continues to execute.

Concurrent collectors can also be implemented in a way that permits the GC to use multiple threads in parallel (even if the GC is implemented as stop-the-world), for instance, as a concurrent mark-sweep style collector [22, 53]. Traditionally, a mark-sweep collector passes over memory twice during a single collection cycle. The first pass locates and marks all reachable allocated objects in the program. The second pass then scans the entire memory area reclaiming the data for all objects that were never marked. The mark and sweep passes can be made parallel such that separate threads can be used to mark and collect different portions of the memory space.

Reference Counting

Reference counting is a technique used by some GC algorithms to determine when an object is no longer reachable. Each object has a counter, and each time the object is referred to during execution (e.g., assignment), its counter is incremented. Each time a reference is removed, its counter is decremented. When the count reaches zero, the object is no longer reachable and its memory can be reclaimed. It must be noted that reference counting is not exclusive to GC. For instance, Gay and Aiken implemented their RBMM system via use of a reference counter. They found that maintaining
a counter can have a large performance impact on certain applications [28].

**Generational Collecting**

In contrast to reference counting, which introduces an overhead of incrementing and decrementing a counter associated to each allocated object when it is referenced, a generational collector groups objects based on their survivability of collection. Memory associated to younger objects is scanned for garbage more frequently than that for older objects. When an object survives a number of collection cycles, it is promoted to an older generation, which is scanned less frequently. This copying compacts the memory space and enhances data locality.

**2.3.3 Memory Footprints**

Figures 2.3 and 2.4 \(^3\) compare the memory footprint of a hypothetical program using GC and RBMM. Figure 2.3 illustrates the difference in memory occupancy, assuming RBMM is based on an ideal program analysis. In this ideal case, the region inference analysis has produced transformations that allocate memory items\(^4\) into regions of sufficient size as to not generate region bloat, while also reclaiming memory at the earliest possible time. In contrast, Figure 2.4 shows that a more conservative RBMM analysis can result in a worse-case scenario for regions, whereby a region becomes increasingly large and acts as a memory leak.

Our motivation is to compare the performance (both space and time) between Go programs using the existing Go GC system versus our RBMM system. An important research question is whether it is possible to achieve a more predictable and consistent footprint as presented in Figure 2.3, for most programs.

\(^3\)These figures are based off of performance results in [69].

\(^4\)Since Go has structured data types (objects), we generically refer to all values resulting from an allocation as an *item*. An item can be a pointer to a primitive (*e.g.*, `var p *int`) or a complex structure/object/aggregate.
2.3.4 Language Influences on Automatic Memory Management

Automatic memory management offers the benefits of safety and convenience to the programmer by reducing their need to manage a program’s memory resources. To make such an enhancement practical for common programming languages to adopt, the performance of such automated management features must not be significantly worse than that of a manually managed system. In addition, the design of a programming language can greatly influence the effectiveness of an automated memory system.

Different languages pose different challenges for RBMM; for example, logic programming languages require RBMM to work in the presence of backtracking. A prominent feature of Go are “go-routines”: independently scheduled threads. Go-routines may share memory, and they may communicate via named channels, à la Hoare’s Communicating Sequential Processes (CSP) [49]. We defer our discussion of implementing RBMM in the presence...
of Go’s concurrency features until Chapter 6, up until that point we focus on a sequential fragment of Go. The sequential part is essentially a safer C extended with many modern features, including higher-order functions, interface types, a map type, array slices, and a novel escape mechanism in the form of “deferred” functions. These features are described in Appendix A.

Typing Influence

How a language exposes data typing to the programmer can influence the overall effectiveness of a system’s management of memory. Data typing is a means for the programmer to tell the compiler what type a specific variable is (e.g., int, char), by providing declarations of program variables. In garbage collected languages, this can have important influences on how a particular object is collected. The type tells the collector what the shape of the object looks like, and at collection-time the collector knows which fields of the object are pointers and which pieces are not (e.g., integers).

Consider a weakly typed language, such as C. Such a language permits opaque pointers; a pointer to a specific blob of memory which can be cast into an object of another type. This makes handling GC tricky, as the GC is not always aware of the shape of an object and thus cannot traverse it properly for collection. While C is not typically garbage collected, there do exist collectors which provide such capabilities [8].

Go is strongly typed; the compiler knows the type of each variable. Since Go is garbage collected, this typing property can prove beneficial in creating a precise (type-accurate) collector. Go does permit type conversion, where specific types can be converted to another type if they have the same underlying type. The details of this feature are not necessary for this discussion, but a simple example is converting between two numerical types (e.g., an integer being represented as a float64). Go also permits empty interfaces, which allow a function to accept any data type as an argument to a function. The runtime system can handle this properly because additional metadata is passed to the function telling it what type is assigned to each argument.
Aside from GC, typing also provides information necessary for the compiler to inform the programmer of any type-related errors detected at compile-time. Such errors arise when the program tries to assign values to variables of non-equivalent types, or trying to pass arguments to a function that do not match the function definition. Languages that are strongly typed can prevent memory access errors. Since an object has a known type, the compiler can issue a warning or error if an object is being accessed in a manner that is not sound. Such a case arises if the program tries to access a field of an object that does not really exist.

Typing also influences the lifetime of a variable. Pointers can escape functions and thus permit dynamically allocated memory to be passed around and exist for an amount of time that cannot be known at compile time. The common primitive types, such as integers, have a known size, and are typically copied between function calls. Go is pass-by-value, therefore all data passed between function calls is copied, including arrays and their contents. Pointer types only have their addresses copied and can contain addresses of objects containing additional pointers. Since Go’s `new` allocator returns an address, these pointers might point to dynamically allocated data.

### Object Lifetime Influence

An object’s lifetime specifies how long its contents can be used. The lifetime of an object extends from the first to the last program point where its content can be used. The definition of object lifetime differs from that of variable lifetime. A variable is considered live if there is a program path from which the variable will be accessed. Liveness is a key property that will be discussed in Section 2.3.5, where we compare static and dynamic analysis for determining when an object can be reclaimed.

Normally, objects declared locally within a function reside on the stack and have scope within that function. When the function terminates and the stack frame is popped, those stack-allocated objects are no longer accessible and therefore become dead. Go is an exception to this rule. In cases where
a stack object is returned, the Go compiler will promote that object to the heap. This promotion can extend the lifetime of an allocation beyond the routine in which the object was declared and allocated.

It is important to draw a distinction between an object’s liveness and reachability. An object is reachable if its contents can be accessed from some variable. Dead objects can be reachable from a live object. For instance, a dangling pointer within a linked-list might still refer to older non-relevant list nodes (possibly reclaimed), while the head of the list might still be relevant and live. Objects that are not reachable are considered dead. However, dead objects do not have to be unreachable.

Dynamically allocated data can have a lifetime that extends beyond the function from which it was allocated. This heap-allocated memory can be passed between functions and can be accessed via globally declared variables. Since the stack does not manage the space for these allocations, they must be handled by the programmer or the language’s runtime system.

In languages that require the programmer to manually manage memory, the allocated data must be explicitly freed to return the unneeded data back to the system. Otherwise, the memory would leak. In languages with automatically managed memory (e.g., GC or some RBMM implementations), the system will automatically determine when the memory can be reclaimed. This reclamation point can occur at a later time during program execution than what a good programmer would have otherwise decided. Reclaiming sooner, rather than later, can reduce the total in-use memory space (footprint) of a program; providing the maximum amount of memory resources for the program to utilize.

Global objects can also point to heap allocated memory. Since globals are always accessible, their lifetime is that of the entire program.

### 2.3.5 Static versus Runtime Analysis

RBMM and GC are both automated solutions that differ in ways that can be used to complement each other. Understanding the differences between
the two systems is useful to better understand our formulated comparison in Chapter 4, as well as understand the reasoning behind combining the two, which is discussed in Chapter 5.

According to Rice’s theorem, any non-trivial semantic property of a program, such as deciding an object’s last use, is undecidable [66]. An object’s last use is an important property since it is what an RBMM inference analysis statically approximates. It is important to realize that reachability (what a garbage collector can decide) does not mean that an object will be used. Therefore, an RBMM static analysis can reclaim memory that a garbage collector might not. To demonstrate that last use is an undecidable problem, we sketch a reduction from the halting problem. Consider the halting problem as being represented as $\langle P, i \rangle$ where $P$ is a program and $i$ is an input to $P$. The question asked is this: Will $P$ halt on given input $i$? We can represent an instance of the last use problem as $\langle P, i, o, p \rangle$, where $P$ represents a program, $i$ the input to $P$, $o$ is an object, and $p$ is a program point in $P$. What we want to know is the following: Will there ever be a use of $o$ after $p$ in $P$? Without loss of generality, assume $P$ has a single exit point (if it does not, a simple transformation will achieve this.) We now introduce a function $T$ which takes as input a halting problem instance $\langle P, i \rangle$ and transforms it into a last use problem instance $\langle P', i, o, p \rangle$. $P'$ is identical to $P$, except it has as its first statement an object allocation, which will be for the freshly named $o$, and a single use of $o$ just before the exit point of $P'$. Clearly $P$ will halt on $i$ if and only if the exit point of $P'$ is reached, that is, if $o$ is used after, say, the entry program point. This way $T$ reduces the halting problem to the last use problem, and we conclude that the latter is undecidable.

Because of this undecidability, a static analysis, such as what RBMM requires, cannot determine exactly how long an object will be needed. It can, however, approximate lifetime. On the other hand, a runtime (dynamic) analysis, as used by GC, can only make decisions about what objects are reachable, based on the current state of the program during execution. This
is a different approximation to lifetime. Either case, static or runtime reasoning, approximation to lifetime is involved.

Conservative approximations, however, are still useful. They can be used to decide, at compile-time, where memory for a variable can be allocated and reclaimed in a safe manner. An RBMM solution can only statically reason about a variable at the program points in a program. GC, on the other hand, can only make decisions about which allocated objects are reachable after it scans its graph of nodes, starting from the root set. RBMM approximates reachability by calculating which objects point to which others. This calculation results in a points-to graph providing the compiler with information about what objects can be reached from what program points. Similarly, a garbage collector will scan memory during runtime and follow pointers to other reachable nodes for the given program state. This scan can only decide what can be reached, and cannot not determine which objects will actually be accessed in the future. This is an important point. Just because an object can be reached does not mean that its contents will be used. Objects that are never used can safely be reclaimed, even if they are reachable.

Figure 2.5 conveys this observation, that memory reachability can be reasoned about both statically and at runtime. However, a static analysis cannot do any better than the runtime analysis. In Figures 2.3 and 2.4, the spikes represent unreachable objects that can be reclaimed. Objects that belong to unreachable memory can be safely reused. A runtime solution can maximize the discovery of unreachable memory, but not necessarily memory that will no longer be used in execution. A compile-time solution is not as effective at discovering what will be unreachable; however, this analysis can be more effective at determining the relevance (further use) of memory. A static analysis can determine when memory will never be used, even if it is reachable. Figure 2.6 illustrates this point. Reclaiming reachable but never used memory (non-relevant memory) can be an advantage for minimizing the amount of memory being used during execution. Khedker et al. [52]
propose a static analysis that takes advantage of this idea. Their analysis sets pointers to `null` if the pointer points to an object that is reachable but not used. This will allow the garbage collector to reclaim the dead item’s memory.

Being able to make decisions about when and if memory will be used at a later program point is one of the benefits of RBMM. A garbage collector cannot guess what can be reclaimed later, and can only make decisions about what can be reclaimed when a GC occurs. Additionally, an RBMM solution has the added benefit of knowing that something might be reachable but never used, something a garbage collector cannot do.
2.3.6 Object Relationships

The responsibility of region inference is to statically determine which objects are allocated from which regions. Properties of objects and their relationships are used to establish which regions they should be allocated from. Knowing these relationships can also benefit GC.

Object Points-to Relation

The region inference portion of an RBMM analysis can define regions as sets of objects based on a points-to relationship between these objects. The reachability of one object from another can be seen as the transitive closure of this points-to relationship. The benefit of this approach is that regions
can be created such that there are no objects which point into other regions. This means that a region can be safely removed without the region having to wait for any objects from external regions to die. This also prevents dangling pointers from being created due to pointers referencing data from other regions. However, this relationship can create large regions. Having more objects in a region increases the probability of region bloat, due to a few reachable objects keeping a region containing many dead objects alive.

This connectivity association has also been utilized in Connectivity-based garbage collectors to improve collection time and memory usage [47].

**Object Lifetime Relation**

To maximize memory utilization, a program must allocate memory for an object at the latest possible time before it is referenced. The memory must also be reclaimed at the earliest possible time after it is last referenced. Knowing the lifetimes of the objects within a region allows regions to be constructed with objects that can die together. This relationship can allow for inter-region pointers. Caution must be taken by the region inference algorithm to ensure that reclaiming a region will not have any effects on pointers into the region that is to be reclaimed. This relationship can also reduce the region bloat problem since regions are not associated by connectivity but by lifetime similarity. A lifetime analysis can produce regions that are created later and die sooner. By placing objects with similar lifetimes into the same region, the analysis can reduce the potential of having long-lived objects keep a region with shorter-lived objects alive. Past research has shown that object connectivity (points to) is useful in predicting object lifetime and such information can benefit automatic memory management [48].

GC systems can be built that make use of object lifetime information. Such systems can reduce the cost of GC by frequently processing objects that are assumed to be garbage, and less frequently processing those that are assumed to have longer lifetimes. The weak generational hypothesis, or infant mortality, is the observation that the most recently allocated objects
have a high probability of also becoming garbage the soonest [80, 50, 51, 43]. This hypothesis is not attributed to one person, but is commonly referenced in the GC literature. This concept forms the basis for generational garbage collectors, whereby pools (called *generations*) of memory are set aside for objects of different lifetimes. The young, or most recently allocated objects, reside in a generation that is more frequently collected than that of older objects. Objects get promoted to the older generation if they survive multiple collections. Generational collectors can reduce the amount of work that a collector has to perform, while also reducing memory footprint [58].

### 2.3.7 Implementation Difficulties

From a programmer’s perspective, using a memory management system appears easy, and this can be seen as a danger. Confident programmers assuming that their code is safe might ignore the proper avenues for testing and verifying the integrity of their programs. Such systems can be quite tricky to use correctly without introducing bugs. With that nugget of information in mind, consider writing a memory management system and all of the pitfalls that can occur. As one can imagine, implementing these systems can be riddled with the same complications as using them.

Garbage collectors are difficult to program, and we can attest to the fact that they are additionally complicated and time-consuming to debug. A collector manipulates a running program’s data in a concealed way such that the running program is not aware that its data is being manipulated. This concealment makes it difficult to pinpoint the exact spot at which an error is created. This is especially true for copying collectors, which require that all reachable references to a copied (or relocated) item have their pointers updated. When the program regains control from the garbage collector, an error introduced by the collector will not always manifest itself immediately. In fact, many collections might pass before a problem introduced in an earlier collection surfaces. The problem itself could be caused by numerous GC errors: incorrectly tracing an item, not copying all or the correct
pieces of an item, or not updating a pointer properly. In Chapter 5 we intro-
duce additional complexity by combining RBMM and our region-aware GC, 
which means that we have two memory management systems to develop and 
combine correctly. Sometimes there is a trade-off between simplicity (and 
consequentially stability) and performance.

This thesis composes two memory management systems that I have writ-
ten under the guidance of my advisers. The first is an RBMM system which 
requires that the resulting program be analyzed and transformed correctly 
so that the user’s program does not crash by fault of the memory system. 
Any incorrect assumption or miscalculation during code analysis or code 
generation can result in a faulty program execution.

The other system I have implemented is a copying garbage collector. The 
collector must be absolutely certain of the data type for a specific object 
before it copies it and updates the object’s pointers. Any bad copying, or 
data misinterpretation, can crash the running program or generate incorrect 
results). Our region-aware garbage collector is a proof of concept allowing us 
to explore the interaction with RBMM. Both of our systems are not without 
their bugs; however, they do permit a handful of test cases to be executed 
allowing us to measure the performance of our systems.
Chapter 3

Implementing RBMM for the Go Programming Language

If we knew what it was we were doing, it would not be called research, would it?

- Albert Einstein

3.1 Introduction

In this chapter we introduce RBMM as an automatic memory management solution that can co-exist with Go’s existing garbage collector. Our goal is to achieve the benefits of an automatic memory management system, whereby the user is removed from making (potentially incorrect) decisions on when to reclaim memory. Additionally, we aim to eliminate, as much as possible, the overhead required by a garbage collector. To do this, we implemented an RBMM system that co-exists with the Go programming language’s runtime garbage collector. The implementation and design are covered in this chapter. We provide a formal correctness statement to our design in the following chapter, Chapter 4. This separation eases the understanding of our approach.

Our design aims to both increase the speed of object reclamation while also decreasing the overall footprint of the program. To accomplish fast reclamation our solution groups objects together into regions, this procedure is known as region inference. During runtime, when all of the objects in the region are no longer needed, a fast reclamation region operation is
executed that makes the memory for the entire region available to fulfill future allocation requests. However, choosing when to reclaim a region is not a trivial property to decide, specifically because it relies on object lifetime analysis which is undecidable [66].

Our proof-of-concept uses a static analysis to first analyze the program, and then to insert runtime calls to our region operations. Since a static analysis cannot decide how long all allocations will be used (consider a global pointer to allocated data), our analysis must make decisions such that the runtime system can reclaim allocated objects when it is safe to do so. Our context insensitive analysis passes regions from callers to callees. If a region is needed later in the caller, the caller will first increment a counter in the region such that the region will not be prematurely reclaimed in the callee, or any other functions the callee executes. The context insensitive approach of our static analysis increases the scalability of our design (see Section 3.6.3).

In order to measure the differences between RBMM and GC, within the context of Go, we must first modify both a Go compiler and introduce our RBMM runtime system. The process of transforming a Go program into one that is RBMM aware starts with a static analysis of the program’s source code. When the analysis completes, our compiler then inserts region operations into the program. These region operations are function calls into a shared object file that are called during program execution. We call this object file our runtime library. When the transformed Go program executes, it will call the RBMM routines in our runtime system to perform any RBMM related memory management. This solution provides a design which combines static analysis, to guide region creation, and runtime bookkeeping, to help control memory reclamation.

The novelty, and main advantage, of our approach is that it greatly limits the amount of re-work that must be done after each change to the program’s source code, making our design more practical (scalable) than existing RBMM systems (see Section 3.6.3). The latter is the result of our flow-insensitive [10, 56, 65] and context-insensitive static analysis. Our solu-
tion also introduces a novel concept, “protection counters,” which prevents a region from being prematurely removed. These runtime counters are an efficient alternative to the more computationally expensive reference counters. Reference counters have been studied in RBMM systems before, such as Gay and Aiken’s C@ and RC [28, 29].

We have implemented our RBMM solution as an extension to the gccgo compiler. Our program analyses and transformations deal with GIMPLE, GCC’s intermediate language, but to make our presentation more accessible, we discuss our methods as if they apply to a Go/GIMPLE hybrid whose syntax we present in Section 3.2. Our prototype implementation discussed in this chapter handles almost all of the first-order sequential fragment of Go. This initial exploration also addresses higher-order functions, which is described later in this chapter. We defer our discussion on handling Go’s concurrency primitives until Chapter 6.

While our modifications are implemented for the gccgo Go compiler, the techniques and information presented in this chapter should be applicable to any statically typed language, with some modifications as necessary.

In Appendix A we discuss the syntax and functionality of the Go language. Go programmers are required to explicitly request dynamic memory via the new routine or its variant make, which are like malloc in C. Unlike other languages, Go allows functions to return references to local variables. To avoid dangling references, the Go compiler automatically detects such occurrences, and transforms the function to explicitly allocate storage on the heap for the variable. Memory is never explicitly freed by the programmer; instead, current Go implementations use GC.

3.2 A Distilled Go/GIMPLE Syntax

Since our analysis and code transformations operate on GCC’s intermediate GIMPLE representation, we must reason about Go in both languages (Go and GIMPLE). To simplify our system’s description, we will use the distilled
\[
\begin{align*}
\text{Prog} & \rightarrow \text{Func}\* \\
\text{Func} & \rightarrow \text{func Fname ( Var\* ) \{ Stmt\* return Var \}} \\
\text{Stmt} & \rightarrow \text{Var} = \text{Var} \\
& | \text{Var} = * \text{Var} \\
& | * \text{Var} = \text{Var} \\
& | \text{Var} = \text{Var} . \text{Sel} \\
& | \text{Var} . \text{Sel} = \text{Var} \\
& | \text{Var} = \text{Var} [ \text{Var} ] \\
& | \text{Var} [ \text{Var} ] = \text{Var} \\
& | \text{Var} = \text{Const} \\
& | \text{Var} = \text{Var} \text{ Op} \text{ Var} \\
& | \text{Var} = \text{new Type} \\
& | \text{Var} = \text{Fname ( Var\* )} \\
& | \text{Var} = \text{go Fname ( Var\* )} \\
& | \text{Var} = \text{recv fromVar} \\
& | \text{send Var onVar} \\
& | \text{if Var \{ Stmt\* \} else \{ Stmt\* \}} \\
& | \text{loop \{ Stmt\* \}} \\
& | \text{break}
\end{align*}
\]

Figure 3.1: General Go/GIMPLE syntax

syntax presented in Figure 3.1. This syntax is general enough to cover the interesting features of Go without requiring the reader to be familiar with how the GCC compiler translates from Go syntax into GIMPLE.

\textit{Prog} represents a Go program in our distilled syntax. A Go program consists of a collection of zero or more functions from the syntactic category \textit{Func}. A function definition consists of a name, \textit{Fname}, and a collection of statements from the syntactic category \textit{Stmt}. Data types are from the syntactic category \textit{Type}. \textit{Var} and \textit{Const} are the syntactic categories of program variables and constants respectively. A selector, \textit{Sel}, represents a field within a structure.

This simplified hybrid syntax reflects the fact that we deal with three-
address code when analyzing and transforming a Go program. We have normalized the syntax in obvious ways, requiring, for example, that selectors, indexing, and binary operations are applied to variables, rather than to arbitrary expressions.

3.3 Design

RBMM systems must annotate every memory allocation operation with the identity of the region that should supply the memory. This permits the allocator to produce memory from the particular region that a variable is associated with. To facilitate, at runtime, the allocation and reclamation of memory from a particular region, the RBMM system must also insert into the program calls to the functions implementing the primitive operations on regions (see Section 2.3.1). An allocation cannot come from a nonexistent region. Since we want to minimize the lifetime of each region, we want to insert code to create a region just before the first allocation operation that refers to that region, and we want to insert code to remove a region just after the last reference to any memory item stored in that region. Figuring out which allocation sites should (or must) use the same regions requires analysis.

Every region must be created and removed. The time taken by these operations is overhead. To reduce these overheads, we want to amortize the cost of the operations on a region over a significant number of items. Having each item stored in its own region would impose unacceptably high overheads, though it would also ensure that its storage is reclaimed as soon as possible. Reclamation of a region simply means returning its list of pages to the freelist. Having all items stored in a single giant region would minimize overheads, but in most cases, it would also ensure that no storage is recovered until the program exits. We aim for a happy medium: many regions, with each region containing many items.

The hardest task of the program analysis needed for RBMM is figuring
out the dependencies between regions. If an item A may contain a pointer to an item B, then the region containing item B cannot be reclaimed until the region containing item A is reclaimed, because doing otherwise would leave those pointers dangling. The set of regions will typically form a directed acyclic graph. In principle, it could form a cyclic graph, but any cycle in the graph represents a set of regions in which no region can be reclaimed before any of the others. Since all the pages in those regions would be reclaimed at the same time, merging the regions into one will yield a program with less overhead. We discuss our solution for the dangling pointer problem in Section 3.6.1. The compiler uses analysis data to transform the program by inserting region operations. These function calls, or region annotations, are responsible for creating and reclaiming regions, as well as allocating memory from a region at runtime.

3.4 Region Types

The regions that our static analysis infers from the input source code are of two types: non-global and global. Non-global regions are created and passed as data down to functions. The Global Region holds data for which our analysis cannot deduce a lifetime, such as data pointed to via global pointers. There is only ever a single instance of this global region per program. Recall that it would be incorrect for our system to try to remove data when its lifetime is undecided, since such a removal might reclaim data that is needed later in execution. In the implementation of our RBMM system discussed in this chapter, we use Go’s mark-sweep garbage collector to manage items with undecided lifetimes (items allocated from the Global Region). The garbage collector can safely reclaim items using a runtime analysis. Since GC is a runtime feature, it can slow down program execution. Naturally, we aim to place as much data as possible into non-global regions.
3.5 Runtime Support

The entire purpose of our static analysis is to transform the program by inserting calls to region operations which will be encountered during runtime. These operations manipulate the regions where allocated memory is produced, these calls are discussed in detail in Section 3.7.

We now introduce some concepts that help explain our runtime support for regions. A region flexipage is a fixed-size, contiguous chunk of memory. For allocations that are larger than a standard region page, we round-up the allocation size to the next multiple of the standard page size (hence the *flexi* prefix used in our terminology), therefore our regions can consist of pages of varying sizes. The default page size we choose for our RBMM system is 4KB, which matches the size used in our development machine’s OS. Since all memory items must be wholly contained within a single region page, handling memory allocations of an unbounded size requires the ability to allocate region pages of an unbounded size. Therefore RBMM systems in practice must support multiple page sizes. The region page has a header containing a link field so that pages can be chained into a linked list. The page header also contains the next free memory word on that page.

Since regions manage the memory from a page, having a pointer in a page to the next free word on that page is redundant. In our updated system, discussed in Chapter 5, we move the latter free memory word into the region header. This reduces the size of region pages by one word.

From the perspective of the runtime system, a region is a linked list of pages. The region header contains bookkeeping information about the region, such as its most recent page that it can allocate memory from. As we explain later, it also includes a protection counter. Region headers are not located on any of the pages used for data. Instead, all region headers are managed on a separate series of pages dedicated to containing region headers. This design decision was simple to implement; however, it makes regions disjoint from their data. This can negatively impact cache performance (*e.g.*, cache misses) when the header and its corresponding data are not
both already in the cache. The address of a region’s header is the region handle, through which it is known to the rest of the system. We refer to a variable that holds a region handle as a region variable. Regions are passed as arguments to functions which might allocate memory for an object created in that function.

Our runtime system maintains a freelist of unused region pages. A newly created region contains a single page of the default size (4KB). As allocations are made using a particular region, the region will be extended as needed, taking pages from the freelist if possible, and chaining them onto the region’s list of pages. Quick reclamation is one key benefit of RBMM over that of manual and garbage collected systems.

3.6 Region Inference

Besides a runtime system implementing the basic region operations, our implementation consists of an analysis that decides which region each pointer-valued variable should be allocated in, and a program transformation that introduces calls to RBMM primitives that puts those decisions into effect. This region inference analysis is implemented as two passes: an intraprocedural pass followed by an interprocedural pass. The intraprocedural analysis is concerned with only assignment statements that involve pointers (including memory allocations), while the interprocedural analysis propagates this information across function calls, and is repeated until a fixed-point is reached. Unlike the Tofte and Talpin solution [78], our analysis does not result in a stack of regions. Instead, our region analysis results in sets of regions consisting of objects that can only reach each other. This prevents dangling pointers caused by objects pointing into other regions and those regions being prematurely reclaimed. Our system disallows pointers into other regions.

The goal of our region inference is to generate a set of analysis items for each function within the program. These items are used to guide our
program transformation later on. Each item is a pair that consists of a program variable identifier and an integer value representing which region the variable belongs to. The latter is called the \textit{region-id}. The variable, in this case, represents an allocated item within the user’s program. Since region inference is a static analysis, and regions will not be generated until runtime, the \textit{region-id} is used to group the allocated items together so that our analysis can generate code that will allocate all variables with the same \textit{region-id} from the same region at runtime.

3.6.1 Preventing Dangling Pointers

Our region inference is designed to prevent dangling references. As we discussed earlier, if an object \( A \) points to an object \( B \), and if \( B \)'s region is reclaimed before \( A \)'s, then any access to \( B \) via \( A \) will result in an invalid memory access, since \( A \)'s pointer to \( B \) would be a dangling reference. We solve the dangling pointer problem by unifying objects based on a points-to analysis. This flow-insensitive analysis forms the basis of our variable \textit{unification} (equivalence constraint) generation, which infers regions without dependencies between regions. In other words, a region cannot contain data that will point to data located in another region. This restriction is simple and safe to implement but it also means that regions can grow relatively large in size. We remove this restriction and introduce a field-sensitive analysis in Chapter 5.

Cherem and Rugina’s approach \cite{10} is similar to Steensgaard’s \cite{72} flow-insensitive solution for unifying variables, since they based their unification constraints on the latter. These constraints form the basis of their points-to graph construction.

Our points-to unification analysis is cruder than Steensgaard’s \cite{72}. Our solution places an object, its fields, and any other objects that it points to into the same set (a region), whereas Steensgaard’s solution generates a graph of pointers and what they can point to. In Chapter 5 we relax our solution and permit the fields of an object to belong to separate regions.
Lattner also utilized a unification based analysis in his pool-allocation research [56]. Our design differs from both Cherem’s and Lattner’s since it is context-insensitive. Our solution unifies objects intraprocedurally. Our interprocedural analysis is only concerned with unifying variables that are made at call sites. Our design eliminates the need to reanalyze the callee due to constraints generated in the caller.

3.6.2 Intraprocedural and Interprocedural Analyses

Our intraprocedural and interprocedural analyses form the basis of our unification algorithm. By determining which variables are associated to which other variables (via a points-to relationship), our analysis can discover groups of variables that we call a region. Our analysis associates with each variable \( v \) in the program (or program variable) its own region variable, which we denote \( R(v) \). If \( v_1 \) is a pointer variable, then \( R(v_1) = r_1 \) means that throughout its lifetime, from its initialization until it goes out of scope, whenever \( v_1 \)'s value is not null, \( v_1 \) will always point into region \( r_1 \).

We even associate a region variable with non-pointer-valued variables. If \( v_2 \) holds a structure or array that contains pointers, then \( R(v_2) = r_2 \) means that all the pointers in \( v_2 \) will always point into region \( r_2 \) when they are not null. If \( v_3 \) is a structure or array that does not contain pointers, or if it is a variable of a non-pointer primitive type such as an integer, then \( R(v_3) = r_3 \) means nothing (evaluates to the true constraint), and affects no decisions. Equalities of this last type are redundant, and our implementation does not generate them, but it is easier to explain our algorithms without the tests required to avoid generating them.

Our analyses build sets of equivalence constraints on these region variables. These sets are the result of unifying program variables based on a points-to (assignment) association. This unification merely associates the right-hand side of an assignment with the same set as the left-hand side. For example, the assignment \( a = b \) would cause our analysis to generate the constraint \( R(a) = R(b) \). If the final constraint set built by our analysis does
not require $R(v_1) = R(v_2)$, then we can and will arrange for the memory allocations building the data structures referred to by $v_1$ and $v_2$ to come from different regions.

Our analyses require every variable to have a globally unique name, so we rename all the variables in the program as needed before beginning the analysis. For convenience, we also rename all the parameters of functions so that parameter $i$ of function $f$ is named $f_i$. If the function returns a value, we generate a new variable named $f_0$ to represent it, and modify all return statements to assign the value to $f_0$ before returning it.

Figure 3.2 defines the functions we use to generate region constraints. The top of this figure gives the types of these functions. In these types, $EqConstrs$ is the set of equivalence constraints on region variables (each constraint is itself a conjunction of primitive equivalences); and $Map$ is the set of mappings from function names to sets of these constraints. $S$, $F$, and $P$ generate constraints for statements, function definitions, and programs respectively. The semantic function $S$ is used to produce a set of equivalence constraints from a given statement and $Map$. $F$ is a semantic function producing a new $Map$ from a given function and $Map$. $P$ is a semantic function producing a $Map$ for all functions in the program.

For most kinds of Go/GIMPLE statements, the constraints we generate depend only on the statement. The most primitive statements are assignments, and since Go/GIMPLE is a form of three-address code, each assignment performs at most one operation, and the operands of operations are all variables.

The assignment $v_1 = v_2$, where $v_1$ and $v_2$ are pointers or structures containing pointers, can refer to (alias) the same memory. In this case we constrain the variables to obtain their memory from the same region. If they are not pointers, this is harmless.

After the assignment $v_1 = *v_2$, $v_2$ points to the region in which $v_1$ is stored. Since $v_2$ can point only into $R(v_2)$, the region in which $v_1$ is stored will be $R(v_2)$. The region that $v_1$ points into, that is, $R(v_1)$, can thus be
Map = Func → EqConstrs

S : Stmt → Map → EqConstrs
F : Func → Map → Map
P : Prog → Map

\[
S[v_1 = v_2]\rho = (R(v_1) = R(v_2))
\]
\[
S[v_1 = *v_2]\rho = S[*v_1 = v_2]\rho = (R(v_1) = R(v_2))
\]
\[
S[v_1 = v_2.s]\rho = (R(v_1) = R(v_2))
\]
\[
S[v_1.s = v_2.s]\rho = (R(v_1) = R(v_2))
\]
\[
S[v_1 = v_2[v_3]]\rho = S[v_1[v_3] = v_2]\rho = (R(v_1) = R(v_2))
\]
\[
S[v = const]\rho = \text{true}
\]
\[
S[v_1 = v_2 \text{ op } v_3]\rho = \text{true}
\]
\[
S[v = \text{ new } t]\rho = \text{true}
\]
\[
S[\text{if } v \text{ then } \{ s_1 \ldots s_n \} \text{ else } \{ t_1 \ldots t_m \}]\rho = (\bigwedge_{i=1}^{n} S[s_i]\rho) \land (\bigwedge_{i=1}^{m} S[t_i]\rho)
\]
\[
S[\text{loop } \{ s_1 \ldots s_n \}]\rho = (\bigwedge_{i=1}^{n} S[s_i]\rho)
\]
\[
S[\text{break}]\rho = \text{true}
\]
\[
S[v_0 = f(v_1 \ldots v_n)]\rho = \theta(\pi_{f_0 \ldots f_n}(\rho(f))
\]
where \(\theta = \{ f_0 \mapsto v_0, \ldots, f_n \mapsto v_n \}\)

\[
F[\text{func } f (f_1 \ldots f_n) \{ s_1 \ldots s_m; \text{ return } f_0 \}]\rho = [f \mapsto (\bigwedge_{i=1}^{m} S[s_i]\rho)]
\]
\[
P[d_1 \ldots d_n] = \text{fix} \left( \bigcup_{i=1}^{n} F[d_i] \right)
\]

Figure 3.2: Region constraint generation
reached from $R(v_2)$. Many RBMM systems handle such assignments by establishing a dependence between $R(v_1)$ and $R(v_2)$ requiring $R(v_2)$ to be reclaimed before $R(v_1)$ (if $R(v_1)$ were reclaimed while $R(v_2)$ is in use, some pointers in $R(v_2)$ could be left dangling). Such restriction can be the case when imposing a stack-of-region ordering to region allocation (See Chapter 2.3.1)[78, 12]. Instead, we simply unify $v_1$ and $v_2$ resulting in both of their data being stored in the same region. This is safe, but overly conservative. We handle all assignments involving pointer dereferencing, field accesses, and array indexing the same way, for the same reason.

Assignments involving constants obviously generate no constraints. Since Go does not support pointer arithmetic, assignments involving arithmetic operations have no implications on memory management. Assignments that allocate new memory also do not impose any new constraints: the region in which the allocation will take place is dictated by the constraints on the target variable, not by any property of the allocation operation itself.

To process a sequence of statements (whether in a function body, in an if-then-else branch, or in a loop body), we simply conjoin the constraints from each statement. We also conjoin the constraints we get from the then-parts and else-parts of if-then-elsees. In Go/GIMPLE, all loops look like infinite loops with break statements inside if-then-elsees. The break statement generates no new constraints. All these rules say that the constraints imposed by the primitive statements must all hold, regardless of how those primitives are composed into bigger pieces of code.

The most interesting statements for our analysis are function calls. (They may or may not return a value; if they do not, we treat them as returning a dummy value, which is ignored.) A function call is the only construct whose processing requires looking at $\rho$, which maps the names of functions to the set of constraints we have generated so far for the named function’s body. That function body may require some of the function’s formal parameters to be in the same region, and when processing the call, we need to impose corresponding constraints on the corresponding actual parameters.
The rule for function calls starts by looking up the name of the callee in $\rho$ (this is what $\rho(f)$ does); this will yield a constraint. This rule then projects with $\pi$ that constraint onto the formal parameters of the callee ($f_0...f_n$), including the one representing the return value ($f_0$). $\pi_{f_0...f_n}$ represents the existential quantification of all the variables in $f$ except those of the formal parameters $f_0...f_n$. Therefore, $\pi$ acts as a constraint filter and discards all the primitive constraints involving variables other than formal parameters, but keeps their implications. For example, given the constraints $R(f_1) = R(v_5) \land R(v_5) = R(f_2)$, the projection $\pi_{f_1,f_2}$ yields $R(f_1) = R(f_2)$. The rule for function calls then renames the program variables inside these constraints to refer to the actual parameters in the caller, not the formal parameters in the callee. For example, if the call had $v_8$ and $v_9$ in the first two argument positions, this renaming would yield $R(v_8) = R(v_9)$.

This process depends on $\rho$ containing the correct constraint for every function in the program. Calculating the constraints for all functions in a program is iterative. Since functions can be mutually recursive our analysis should only terminate when no future updates to the constraint set can be made. The $fix$ function is used to calculate the least-fixed point solution to a function $\bigcup \mathcal{F}[d_i]$ in the program. Each function definition, denoted by $d_i$, contributes its constraint to $\rho$. $\bigcup$ is used to join all of the individual function definition constraints together in $\rho$.

For $\mathcal{F}$, we begin our analysis with $\rho$ mapping the name of every function to $true$, reflecting that we do not yet have any constraints about any of the program’s functions. We compute a new $\rho$ reflecting the constraints each function would impose if none of the functions it calls imposed constraints (our intraprocedural analysis). For our interprocedural analysis, we repeat this computation, beginning each iteration with the $\rho$ just computed, until the analysis reaches a least fixed-point (when the resulting $\rho$ is the same as it was in the previous iteration).

Figure 3.3 is an example program from which our analysis produces the following constraints. (Some additional constraints will occur for temporary
package main

type Node struct {id int; next *Node;}

func CreateNode(id int) *Node {
    n := new(Node)
    n.id = id
    return n
}

func BuildList(head *Node, num int) {
    n := head
    for i:=0; i<num; i++ {
        n.next = CreateNode(i)
        n = n.next
    }
}

func main() {
    head := new(Node)
    BuildList(head, 1000)
    n := head
    for i:=0; i<1000; i++ {
        n = n.next
    }
}

Figure 3.3: Creating a linked list in Go

variables introduced in the GIMPLE code, but we ignore those here):

- **CreateNode**: $R(\text{CreateNode}_0) = R(n)$,

- **BuildList**: $R(n) = R(\text{BuildList}_1) \land R(\text{CreateNode}_0) = R(n)$

- **main**: $R(n) = R(\text{head})$. 
package main

type Node struct {id int; next *Node;}

func CreateNode(id int, reg *Region) *Node {
    n := AllocFromRegion(reg, sizeof(Node))
    n.id = id
    RemoveRegion(reg)
    return n
}

func BuildList(head *Node, num int, reg *Region) {
    n := head
    for i:=0; i<num; i++ {
        IncrProtection(reg)
        n.next = CreateNode(i, reg)
        DecrProtection(reg)
        n = n.next
    }
    RemoveRegion(reg)
}

func main() {
    reg1 := CreateRegion()
    head := AllocFromRegion(reg1, sizeof(Node))
    IncrProtection(reg1)
    BuildList(head, 1000, reg1)
    DecrProtection(reg1)
    n := head
    for i:=0; i<1000; i++ {
        n = n.next
    }
    RemoveRegion(reg1)
}
3.6.3 Scalability

This is inherently a whole-program analysis, and that threatens to make it impractical for real use. Therefore, we have designed our analysis to permit practical implementation. First, the analysis is flow and path-insensitive, since the order in which statements in a function body are executed, and which branch of a conditional will be executed, are not significant. This helps make the analysis scalable. More importantly, and contrary to most existing RBMM implementations that we know of, our analysis is context (or call) insensitive: the analysis of a function depends only on the functions it calls, not on the functions that call it. When program transformations depend upon a whole-program context sensitive analysis, a change anywhere may require reanalyzing and recompiling any part of the program. With a context-insensitive analysis, only modules that import a changed module will need to be reanalyzed and recompiled, and only when the analysis result for an exported function has actually changed. We believe this will reduce the need for reanalysis and recompilation to the point that this approach will be practical.

3.7 Transformation

Once the program analysis is complete, we transform the program to use region-based primitives for memory management. This involves replacing calls to Go’s memory allocation primitives with those of our RBMM memory allocator, and inserting calls to create and remove regions. To support this functionality, we must also transform functions to take regions as input arguments. This transformation generates a function that is region polymorphic [78, 30].

The following region primitive operations are inserted into the program to implement RBMM:

- *CreateRegion()*: Create an empty region from which memory items
can be allocated. This transformation occurs when our analysis determines that a future allocation will happen and a region for that allocation is needed. This operation must occur prior to a call to `AllocFromRegion()`.

- **AllocFromRegion(r, n)**: Allocate `n` bytes from region `r`. This function replaces calls to Go’s `new` and `make` functions.

- **RemoveRegion(r)**: Reclaim the memory of region `r` so that it can be reused later, if the region’s protection count and thread reference count are both zero. Our analysis performs a check on program points intraprocedurally to determine where to attempt to reclaim memory managed by `r`. Ideally, this program point should be at the soonest point where objects in `r` are no longer reachable.

- **IncrProtection(r)**: Increment the region’s protection count, ensuring that calls to `RemoveRegion(r)` do not actually reclaim `r` until after `DecrProtection(r)` is called. We explain the role of this operation in Section 3.8.

- **DecrProtection(r)**: Decrement the region’s protection count.

As discussed in Section 3.6, our analysis only summarizes the region equivalence constraints imposed by each function and the functions it calls; it does not collect the region constraints imposed by the callers of each function. This means that some callers to a function may require a certain region parameter to survive the call to the function, while others do not. To minimize memory usage, we want to reclaim the region at the earliest point in the program. This point might be in a callee when the caller no longer needs the region. Therefore, we introduce region protection counts and distinguish between *reclaiming* a region, which actually deallocates the storage, and *removing* a region, which reclaims the region if and only if its protection count is zero. Thus each function is expected to remove the regions associated with its input parameters, (but not those associated with
its return value) as soon as it is finished with them. When a region passed
to a function is needed after the function call, we increment the protec-
tion counter for the region before the call, and decrement it after the call.
This small runtime overhead is the price we pay for limiting ourselves to
a context-insensitive program analysis. Figure 3.4 shows the automatically
transformed version of the code in Figure 3.3.

We present the transformation of program fragment $Syn_1$ into $Syn_2$ using
the notation:

\[
Syn_1 \rightsquigarrow Syn_2
\]

Transformations may be applied in any order, and we apply them repeatedly
as long as any of them are applicable.

We use a few auxiliary functions to access the analysis results for pro-
gram $P$. $\text{compress}_f(r_0, r_1, \ldots r_n)$ is the list of regions $\langle r_0, \ldots r_n \rangle$, without
duplicates, as implied by the region equivalence constraints for $f$’s formal
parameters $(f_1, \ldots f_n)$ and return value $(f_0)$. $\text{reg}(f)$ is the set of all distinct
regions needed for the definition of function $f$, as determined by $\mathcal{P}(P)(f)$.
$\text{ir}(f)$ is the set of distinct regions of the parameters of function $f$, that is
\[\text{ir}(f) = \text{compress}_f(\mathcal{R}(f_0), \mathcal{R}(f_1), \ldots \mathcal{R}(f_n)).\] (Since these regions are given to $f$ by its caller, they are $f$’s input regions.) $\text{used}(S_1; \ldots S_n)$ is the set of
regions used by any of the statements $S_1; \ldots S_n$. $\text{nonlocal}(S)$ is the set of
regions used for variables appearing in statement $S$ other than for variables
scoped to $S$ or some statement within $S$. That is, it is the set of regions
used within $S$ that are used by another statement later and must outlive $S$.

### 3.7.1 Region-Based Allocation

We must replace all uses of Go’s `new` or `make` primitives with calls to our
special region allocator, $\text{AllocFromRegion}(r, n)$. This primitive requests $n$
bytes of dynamic data from region $r$.

\[
\begin{array}{c}
v := \text{new } \langle t \rangle \\
\rightsquigarrow \\
v := \text{AllocFromRegion}(\mathcal{R}(v), \text{size}(t))
\end{array}
\]
Recall that objects with undetermined lifetimes are stored in the Global Region and thus managed by Go’s garbage collector.

### 3.7.2 Function Calls and Declarations

Every function that takes pointers (or structures containing pointers) as input or returns them as output must be transformed to also expect region arguments. Recall that the region argument $r_0$ represents allocations that are made for the return value of the callee. We indicate the region arguments of a function by enclosing them in angle brackets following the ordinary function arguments:

$$f(a_1, \ldots, a_m)(r_0, r_1, \ldots, r_n)$$

We use this notation for clarity; our implementation handles region arguments the same way as other arguments.

The transformation must add a region parameter for each function parameter and return value if they are pointers or structures containing pointers. However, if the analysis has determined that the regions of two or more parameters must be equal, only the first must be added.

If our analysis were implemented as a total global analysis, then our system would only need to pass region variables in the cases that the we know a region is required. Instead, our analysis will pass a region for a variable if the variable is ever passed down to another caller in the callee (including an allocation), or used as a return value. This strategy is more efficient than always passing a region even when the caller does not use it, but less efficient than what a total global analysis could provide.

This permits us to transform function definitions to introduce region
parameters:

\[
\begin{align*}
\text{func } f(f_1,\ldots,f_n) & \{ \\
S_1; & \ldots S_m; \\
\text{return } f_0; \\
\} \\
\end{align*}
\]

\[
\begin{align*}
\text{func } f(f_1,\ldots,f_n)(r_0,r_1,\ldots,r_p) & \{ \\
S_1; & \ldots S_m; \\
\text{return } f_0; \\
\} \\
\end{align*}
\]

where \((r_0,\ldots,r_p) = \text{ir}(f)\)

This adds a region parameter for each function parameter, but excludes any that the analysis pass has determined must be equal to the region for a parameter appearing earlier in the parameter list. A corresponding transformation introduces region arguments into function calls:

\[
\begin{align*}
\text{func } f(f_1,\ldots,f_n) \{ \\
S_1; & \ldots S_m; \\
\text{return } f_0; \\
\} \\
\end{align*}
\]

\[
\begin{align*}
\text{func } f(f_1,\ldots,f_n)(r_0,r_1,\ldots,r_p) & \{ \\
S_1; & \ldots S_m; \\
\text{return } f_0; \\
\} \\
\end{align*}
\]

where \((r_0,\ldots,r_p) = \text{compress}_f(R(v_0),R(v_1),\ldots,R(v_n))\)

This transformation also adds a region argument for each function argument, using the analysis of the function being called to compress out (remove) redundant regions. The appropriate region to pass for each argument, and for the return value, is determined by the analysis.

### 3.7.3 Region Creation and Removal

Any region that is created within a function \(F\) and not associated to \(F\)'s return value or formal parameters, is considered local. In such a case \(F\) is responsible for reclaiming memory of the region before returning back to \(F\)'s caller. The transformation pass tries to create regions at the latest possible time, and remove them as early as possible. There are two ways a function may obtain a region: it may receive the region from its callers, or it may create the region itself. Conversely, there are three ways a function may complete if it has a region: it may explicitly remove the region, it may pass
the region to a function that is responsible for removing it, or in the case of the region associated with the function’s return value, it may allow the region to remain after the function completes execution. This is handled by the following transformations. Of course, the simplest case is when the function creates a region, but does not pass it down to any callee functions. Intuitively we call these local regions, as the region never escapes from the function which created it.

\[
\begin{align*}
\text{func } f(f_1, \ldots, f_n) \{ \\
S_1; \ldots S_m; \\
\text{return } f_0; \\
\}
\end{align*}
\]

\[
\begin{align*}
\text{func } f(f_1, \ldots, f_n) \{ \\
C; S_1; \ldots S_m; R; \\
\text{return } f_0; \\
\}
\end{align*}
\]

where \( C = \{r=\text{CreateRegion}(); \mid r \in \text{reg}(f) \setminus \text{ir}(f)\} \)

\( R = \{\text{RemoveRegion}(r); \mid r \in \text{reg}(f) \setminus \{\text{R}(f_0)\}\} \)

This places all the needed allocations at the beginning of each function body, and all required region removals at the end. The next two transformations migrate those primitives to their best location in the function body. We currently insert region creation operations at the program points just before the first allocation is requested from that region. (Note that even though we do not currently migrate the region creation call for the implementation discussed in this chapter, we still discuss potential transformations that can benefit overall performance.)

\[
\begin{align*}
\text{r=CreateRegion}(); \\
S_1; \ldots S_m; \\
S_{m+1}; \ldots S_n;
\end{align*}
\]

\[
\begin{align*}
S_1; \ldots S_m; \\
r=\text{CreateRegion}(); \\
S_{m+1}; \ldots S_n;
\end{align*}
\]

where \( r \notin \text{used}(S_1; \ldots S_m) \)
For convenience, our implementation actually places the removal at the end of the basic block that contains the statement of last use for that region.

Two more transformations allow region creation and removal to migrate into loops and conditionals. Moving region creation and removal into a loop adds runtime overhead, but by reclaiming memory earlier, it may significantly reduce peak memory consumption. Since the compiler cannot determine whether the amount of memory that will be allocated across a loop can lead to out-of-memory errors, region creation and removals can be pushed (as a pair) into loops where possible. We could also push region creation and removal into conditionals where possible, because it can reduce peak memory use.

Our current system does move the region removal calls if they are inside a loop. If the creation is outside of a loop, then the removal is moved just outside of the loop. This prevents the dangling case where the loop will reuse a region that was removed in a previous iteration. If the region creation is inside a loop, then the removal is placed as the final statement in the basic block before the loop jumps back to the loop start. This allows the memory to be reclaimed per loop iteration.
The next case discussed here applies when a region is only used within the `then` branch of a conditional. Earlier transformations will have moved the region creation before and the removal following the conditional statement.

The same case occurs for the `else` branch of a conditional. If a region is used within a branch, an initial conservative approach is to place the creation prior to the branch, and if $r$ is not used after the conditional, then the `RemoveRegion()` operation can be placed immediately following the conditional.
be placed in that branch. If this optimization were not applied, and if the *then* condition is rarely executed, then any conditions satisfying the *else* case would waste cycles allocating a region that is never used.

\[
\begin{align*}
    & r = \text{CreateRegion}(); \\
    & \text{if } t \{ \\
    & \quad S_1; \ldots S_m; \\
    & \} \text{ else } \\
    & \quad S_{m+1}; \ldots S_n; \\
    & \} \\
    & \text{RemoveRegion}(r); \\
\end{align*}
\]

\[
\begin{align*}
    & \text{if } t \{ \\
    & \quad S_1; \ldots S_m; \\
    & \} \text{ else } \\
    & \quad r = \text{CreateRegion}(); \\
    & \quad S_{m+1}; \ldots S_n; \\
    & \quad \text{RemoveRegion}(r); \\
    & \} \\
\end{align*}
\]

where \( r \not\in \text{used}(S_1; \ldots S_m) \)

Similarly, if \( r \) is only used within the *else* branch of the conditional, then both the CreateRegion() and RemoveRegion() operations can be pushed into that branch. As with the previous optimization, this optimization serves to reduce unnecessary region operation calls. Additionally, moving the CreateRegion() to the latest program point where it will be used and the RemoveRegion() to the soonest point where the region is no longer used makes for a more optimal use of the program’s footprint.

Another optimization can be performed when the analysis results in a CreateRegion() followed immediately by a RemoveRegion(); we can remove
both calls:

\[
S_1; \ldots S_m; \\
r = \text{CreateRegion}(); \\
\text{RemoveRegion}(r); \\
S_{m+1}; \ldots S_n; \\
\Rightarrow S_1; \ldots S_m; \\
S_{m+1}; \ldots S_n;
\]

where \( r \notin \text{used}(S_{m+1}; \ldots S_n) \)

### 3.8 Region Protection Counting

We believe that our introduction of a per-region protection counter makes our RBMM solution different from other RBMM systems. To remove each region at the earliest possible time, we must put a call `RemoveRegion(r)` immediately after the last use of any object stored in region \( r \). To determine even a conservative approximation of the earliest place each region can be removed requires a global analysis of the program. This is difficult to implement, and doubly so to implement incrementally, so that after a small change to a program, only the functions that need to be reanalyzed will be. Instead, we opted for a simpler analysis. Our analysis processes the modules of the program, and the functions in each module, bottom-up (analyzing callees before callers, and analyzing mutually recursive functions together). This is simple to implement and allows efficient compilation, but does not permit the code generated for a function to be influenced by call contexts.

When compiling a function, we cannot know whether or not it should remove the regions it uses; that depends on the call path to that function (that is, the call stack at the time the function is called). The ideal way to allow the caller to determine which regions are removed is to have a specialized version of each function for each combination of regions it should free.
However, this can generate exponentially many versions of each function, and may greatly increase the size of the executable, reducing instruction cache effectiveness.

Another alternative would be for each function to remove only the regions that all its callers agree should be removed, and for callers of that function that require any other region to be removed to remove it themselves after the call. However, by delaying region removal, this may increase peak memory consumption, possibly to an unacceptable level.

We have implemented a third approach: dynamic protection counts. With this approach, each region maintains a protection count of the number of frames on the call stack that need that region still to exist when they continue execution. We transform each function to remove all regions passed to it as arguments, except the region for the return value, provided their protection count is zero. We also transform the function body so that for each region $r$ that is passed in a function call, if any variable $v$ with $R(v) = r$ is needed after the call, we invoke $\text{IncrProtection}(r)$ before the call, and we invoke $\text{DecrProtection}(r)$ after the call:

\[
\begin{align*}
S_1; \ldots S_m; \\
v = f(\ldots)\langle\ldots, r, \ldots\rangle \\
S_{m+1}; \ldots S_n;
\end{align*}
\]  
\[\sim\]  
\[
\begin{align*}
S_1; \ldots S_m; \\
\text{IncrProtection}(r); \\
v = f(\ldots)\langle\ldots, r, \ldots\rangle \\
\text{DecrProtection}(r); \\
S_{m+1}; \ldots S_n;
\end{align*}
\]

where $r \in \text{used}(S_{m+1}; \ldots S_n)$

However, if $r$ is not needed after the call, we do not do this transformation. This ensures that if a function $f$ is called with a region $r$ in a state that would allow it to be removed, and if the last use of $r$ in $f$ is in a call to $g$, $g$ will be called in a state that would allow $r$ to be removed.

One potential future enhancement is to add an additional transforma-
tion to remove unnecessary calls to \texttt{IncrProtection(}r\texttt{)} and \texttt{DecrProtection(}r\texttt{)}, leaving only the first increment and last decrement.

\begin{align*}
S_1; \ldots S_m; \\
\texttt{DecrProtection(}r\texttt{)}; \\
S_{m+1}; \ldots S_n; \\
\texttt{IncrProtection(}r\texttt{)}; \\
S_{n+1}; \ldots S_q;
\end{align*}

\begin{align*}
S_1; \ldots S_m; \\
S_{m+1}; \ldots S_n; \\
S_{n+1}; \ldots S_q;
\end{align*}

where \( r \in \texttt{used}(S_{n+1}; \ldots S_q) \)

Another potential future enhancement is to implement an additional analysis pass that will collect, for each call to each function, information about the protection state of each region involved in the call. Specifically, we want to know whether its maximum protection count at the time of the call is zero, and whether its minimum protection count is at least one. If we have this information about all calls to a function, then we can optimize away either the function’s remove operations on a region (if all the callers need the region after the call) or the “test of the protection count” inside those remove operations (if none of the callers need the region after the call). If the calls disagree about whether they need a region after the call or not, we can also create specialized versions of the function for some call sites, preferably the ones which are performance critical.

It is important to note that a region’s protection count indicates the number of \textit{stack frames} that refer to the region. We modify this counter only twice per function call: once to increment it and once to decrement it. This is in contrast to \textit{reference} counts, which count the number of individual pointers to an item or region. For example, in RC [29], a region-based dialect of C, reference counts must be updated for each pointer assignment. To our knowledge, protection counting is unique to our approach, and avoids the overhead of incrementing/decrementing a per-item counter as well as the
space required for storing the item’s associated counter.

3.9 Higher-Order Functions

Up until this point of the thesis our implementation focuses on the first-order subset of Go. We now consider the higher-order aspects of the language in terms of our proof-of-concept.

Go supports the passing of functions as arguments to other functions, in this case the compiler generates code that passes around the address of the functions. This higher-order feature forms the basis of Go’s ability to handle closures. For instance, go-routines and deferred functions can be more formally thought of as closures. Recall that our analysis and transformations operate on GIMPLE intermediate representation. By the time our analysis begins, GCC will have already transformed `defer` and `go` statements into special cases of higher-order functions (closures and function pointers). Closures are presented to our analysis as function calls to a function that has access to state information. The GIMPLE representation allows our analysis to process higher-order constructs in a more generalized context than having to deal with `defer` and `go` statements specifically.

Higher-order function calls present our transformation with a tricky case, because in general we cannot determine what function will actually be called, so we cannot immediately determine what regions should be passed in the transformed function call. For instance, consider a program which has many functions with a single formal parameter of a pointer type. Our analysis will transform a subset of these functions, because their input arguments are associated with regions. Now we have a case where the function signatures might differ for the regions that have been transformed:

\[
\begin{align*}
  f(v_1) & \rightsquigarrow f(v_1)(r_1)
\end{align*}
\]

In this example, the signature remains the same on the unmodified functions:
\[
\begin{align*}
\text{f}(v_1) & \rightsquigarrow \text{f}(v_1)() \\
\end{align*}
\]

In other words, some functions might be unaltered and still have a single input argument, while other functions might have two. This means that some single pointer argument functions do not match those of the transformed functions. Our static analysis cannot always decide what function will be assigned to a variable, the original version of the function or the version transformed to take region arguments.

Our analysis currently handles such cases by first locating when a function is being assigned to a variable during the interprocedural analysis pass. If that function (a closure) requires region arguments, then we insert a \textit{trampoline} function at the assignment site instead of assigning the original closure. We use the term “trampoline” to refer to compile-time created functions which are responsible for mapping arguments and their associated region arguments to a closure call. This occurs for both function call arguments and for assignments to function variables:

\[
\begin{align*}
g := \text{f}(v_1) & \rightsquigarrow g := tr(r_1)\{f\} \\
\end{align*}
\]

The trampoline, \textit{tr}, takes exactly one region variable argument for each argument the original closure being replaced had, if that original argument had a type that could possibly need an associated region (for example, a pointer). The trampoline also takes another argument for the return value of the closure being replaced, if the return type might possibly be associated with a variable that needs a region. The body of the trampoline wraps the original closure that would have been assigned, and maps the input arguments of the trampoline to the input arguments of the original closure.
3.10 Interface Types

For a datatype in Go to be an interface type, that datatype must have a set of methods matching the function prototypes declared by a particular interface definition. Such functionality is implemented by the GCC Go front-end through the use of an interface method table. Each instance of an object, which belongs to an interface, contains a pointer to a table of methods. The table contains a pointer to each method satisfying the interface type. Our implementation discussed in this chapter allocates all types with an interface method type from the global region.

3.11 Map Datatype

The map datatype was originally managed by a series of runtime functions provided by Go. These functions do not take into consideration regions that our system inserts, and which a map might have been allocated from. Since maps are a built-in datatype, we added the original map functions provided by the Go runtime system, and then modified them to be region-aware. During code transformation, calls to the original map functions are replaced with the region-aware versions. This modified functionality allows for the map datatype to be allocated from regions, and not from the global region.

3.12 Returning Local Variables

The Go front-end handles the return of local variables by allocating memory dynamically and returning the address safely. Since the front-end has already transformed such cases, our analysis handles this situation trivially. We simply replace the allocation call with our region allocator and transform the function as we would any other function that returns dynamically allocated data.
3.13 Region Merging

The concept of combining two regions into a single region, known as *region merging* can be seen as both an optimization and necessity. Runtime merging is necessary in the case of higher-order functions. For instance, the example provided in Figure 3.5 will require that $R(a) \equiv R(b)$. In the case of Figure 3.5 our static analysis cannot determine what function will be called, since $fn$ could represent many functions. Therefore the decision to merge must happen at runtime. $fn$ might point to $foo$ where it will require that $R(a) \equiv R(b)$. For other closures passed as an argument, it is possible that $R(a) \neq R(b)$. In this case we must dynamically merge $R(a)$ and $R(b)$ at runtime if $fn$ is a pointer to $foo$. If we were not to do this, a dangling situation could arise if $R(b)$ were to be removed before $R(a)$. In this case our analysis will perform the merge within the trampoline function that $fn$ calls. Trampolines are described in Section 3.9.

One problem with runtime merging is that a program might have two regions that must be merged, where one of them is the global region. Merging anything with the global region will not make the program incorrect, but is
inefficient since it puts memory allocated from our RBMM region allocator into a region whose data are garbage collected. Since the garbage collector never allocated the region data, it cannot remove it. Therefore, any regions that merge with the global region become memory leaks and will never have their memory reclaimed.

Another case when merging is a necessity happens at compile-time. These merges occur when our static analysis detects that two formal parameters for a function become associated, as when a member of one is assigned to a member of another (region aliasing). In such a case, our analysis will combine the two regions into one during compile-time, avoiding the overhead of making a runtime call to combine two regions.

Our transformation modifies the function to take one single region for both parameters, and thus merging occurs seamlessly and without any runtime overhead:

\[
\begin{align*}
  f(e_1, e_2) & \leadsto f(e_1, e_2) \langle r_1, r_2 \rangle
\end{align*}
\]

Merging can also be used as an optimization. For example, Figure 3.6 is a case where an analysis can detect at compile-time where a merge might occur. This occurs when a function call occurs within a branching statement. Since the branch might be taken infrequently or never, we want to avoid the case of generating one large region from multiple smaller ones. Therefore, in this case we can defer the decision to merge until runtime.

In Figure 3.6, the merge operation should be inserted before the call to \textit{bar} in the \texttt{then} branch of the \texttt{if} conditional. This is less conservative than the compile-time merge which will generate a merged region no-matter-what. We detect the conditional case similar to the compile-time merge mentioned above. A merge function is inserted into the code in the branch just before the function call, that requires multiple arguments to be from the same region, is called. This merge function will be executed during runtime. Unfortunately, runtime merging is not without some performance overhead. It requires additional checks at region removal, as well as the actual merge
function responsible for associating the regions together.

Our concept of merging is safe, although not optimal. Arbitrarily merging two regions together is safe, since the memory for the objects of the merged region can never be reclaimed until all of the objects are no longer needed. A program will hold the same semantics if two regions unnecessarily become one; however, the memory utilization of the program will be worse.

### 3.14 Multiple Specialization

Caution must be taken when analyzing function calls that belong to libraries not compiled by a region-aware compiler. Since our modified compiler (gccgo plugin) might not have access to the library source code, we cannot recompile the library to make it region enabled. Any allocations in such external libraries will come from Go’s garbage collector. Consider the case where an RBMM allocated structure is passed as a pointer to an external function in a non-region-aware library. If this library assigns a Go collector allocated field to this argument, then the field’s memory might be immediately reclaimed during Go’s next GC cycle. This occurs since

```go
func bar(x, y *T) {
    x.next = y
}

func foo(val int) *T {
    a := new(T)
    b := new(T)

    if val == 42 {
        bar(a, b)
    }

    return a
}
```

Figure 3.6: Runtime merging case
the only access to the field might be from the RBMM allocated structure which is outside of the memory space of Go’s garbage collector. Therefore, Go’s garbage collector will never be able to reach the allocated field from a root-set variable during a memory scan, and incorrectly concludes that its memory can be reclaimed. To avoid this case, any arguments we pass, or receive as a result, to an external library must be allocated from the global region (which uses Go’s garbage collector to allocate data). In addition, we cannot pass region arguments to external libraries since we cannot transform or specialize any functions in these pre-compiled binaries.

Now, consider the case when a closure is passed to a function that is part of another Go library or module that was not compiled with a region-aware compiler. If this external library does not have region-aware source code, our transformation cannot pass a region-specific closure or datatype. Such cases require that the functions in the external library file are aware of region semantics, and know how to pass region data to the closure input argument. This is information which these non-region-aware object files do not have. To avoid passing region-aware data to non-region-aware code, we specialize the closure argument. The specialized copy of the closure is never transformed by our region analyses, therefore all allocations will come from the garbage collector. In the case that the external object file was compiled with a region-aware compiler, we can pass region-specific data, such as the trampoline for the closure as mentioned in Section 3.9.

### 3.15 Go Infrastructure

Go programs can consist of multiple object files. These object files form the basis of libraries, or packages, for Go. These files may or may not have been compiled using a region-aware compiler. In the case where the foreign package was not compiled with a region-aware compiler, any information which might contain region data, such as functions or instances of complex objects (structures or interface instances), cannot be passed to the external
object file. The reason for this restriction is that for closures, the non-region-aware libraries will not pass region arguments to the closures, and the body of the closure might expect region arguments. For complex objects with pointer members, the non-region-aware libraries might allocate data from the garbage collector to the member. In this case the garbage collector might not "see" the encompassing structure since it might have been allocated using our region memory. In this case, there is nothing pointing to the member to make it reachable from the garbage collector’s perspective. Therefore, a subsequent GC might reclaim this memory early, regardless that a region expects the memory associated to that member to be valid. In the case we do compile these packages and create our own Go object file, our transformation can pass region-aware data. During our transformation our analysis data is added as another ELF section as part of the object file that the gccgo compiler creates. The additional information denotes the region-modified functions and which arguments and region arguments are associated with that function. During the interprocedural analysis pass, we can query this information and pass arguments and their associated regions to functions in the external object file.

3.16 Evaluation

To test the effectiveness of our implementation, we benchmarked a suite of small Go programs. (We cannot yet test larger programs due to our as yet incomplete coverage of Go.) The benchmark machine was a Dell Optiplex 990 PC with a quad-core 3.4 GHz Intel i7-2600 CPU and 8 GB of RAM, running Ubuntu 11.10, Linux kernel version 3.0.0-17-generic. We used GCC 4.6.3 to run our plugin and compile the benchmarks, but linked with GCC 4.6.1 libraries supplied with the operating system.

Table 3.1 has some background information about our benchmark programs. Some of these are adaptations of Debian’s “Computer Language Benchmarks Game” provided by the GCC 4.6.0 Go test suite and aimed at
<table>
<thead>
<tr>
<th>Benchmark</th>
<th>LOC</th>
<th>Repeat</th>
<th>Alloc</th>
<th>Mem</th>
<th>Collections</th>
<th>Regions</th>
<th>Alloc%</th>
<th>Mem%</th>
</tr>
</thead>
<tbody>
<tr>
<td>binary-tree-freelist</td>
<td>84</td>
<td>1</td>
<td>270</td>
<td>227M</td>
<td>3</td>
<td>1</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td>gocask</td>
<td>110</td>
<td>100k</td>
<td>56M</td>
<td>3.8G</td>
<td>97k</td>
<td>700,001</td>
<td>0.5%</td>
<td>0.1%</td>
</tr>
<tr>
<td>password_hash</td>
<td>47</td>
<td>1k</td>
<td>160M</td>
<td>13G</td>
<td>145k</td>
<td>5,001</td>
<td>~0%</td>
<td>~0%</td>
</tr>
<tr>
<td>pbkdf2</td>
<td>95</td>
<td>1k</td>
<td>115M</td>
<td>8G</td>
<td>92k</td>
<td>12,001</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td>blas_d</td>
<td>336</td>
<td>10k</td>
<td>6M</td>
<td>890M</td>
<td>11k</td>
<td>57,0001</td>
<td>9.2%</td>
<td>9.1%</td>
</tr>
<tr>
<td>blas_s</td>
<td>374</td>
<td>100</td>
<td>49K</td>
<td>5M</td>
<td>58</td>
<td>5,001</td>
<td>10.1%</td>
<td>21.0%</td>
</tr>
<tr>
<td>binary-tree</td>
<td>52</td>
<td>1</td>
<td>607M</td>
<td>19G</td>
<td>282</td>
<td>2,796,195</td>
<td>~100%</td>
<td>~100%</td>
</tr>
<tr>
<td>matmul_v1</td>
<td>55</td>
<td>1</td>
<td>6K</td>
<td>72M</td>
<td>10</td>
<td>4</td>
<td>96.0%</td>
<td>99.9%</td>
</tr>
<tr>
<td>meteor-contest</td>
<td>482</td>
<td>1k</td>
<td>3M</td>
<td>165M</td>
<td>2k</td>
<td>3,459,001</td>
<td>~100%</td>
<td>~100%</td>
</tr>
<tr>
<td>sudoku_v1</td>
<td>149</td>
<td>1</td>
<td>40K</td>
<td>12M</td>
<td>110</td>
<td>40,003</td>
<td>98.8%</td>
<td>99.2%</td>
</tr>
</tbody>
</table>

Table 3.1: Information about our benchmark programs

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>MaxRSS (megabytes)</th>
<th>Time (secs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>GC</td>
<td>RBMM</td>
</tr>
<tr>
<td>binary-tree-freelist</td>
<td>891.84</td>
<td>12.4</td>
</tr>
<tr>
<td>gocask</td>
<td>27.45</td>
<td>71.6</td>
</tr>
<tr>
<td>password_hash</td>
<td>26.60</td>
<td>119.0</td>
</tr>
<tr>
<td>pbkdf2</td>
<td>26.37</td>
<td>71.4</td>
</tr>
<tr>
<td>blas_d</td>
<td>25.87</td>
<td>5.4</td>
</tr>
<tr>
<td>blas_s</td>
<td>26.05</td>
<td>12.2</td>
</tr>
<tr>
<td>binary-tree</td>
<td>1323.74</td>
<td>79.2</td>
</tr>
<tr>
<td>matmul_v1</td>
<td>313.03</td>
<td>11.7</td>
</tr>
<tr>
<td>meteor-contest</td>
<td>27.41</td>
<td>11.0</td>
</tr>
<tr>
<td>sudoku_v1</td>
<td>26.96</td>
<td>15.6</td>
</tr>
</tbody>
</table>

Table 3.2: Benchmark results

measuring language performance (binary-tree, binary-tree-freelist, meteor-contest). The matmul_v1 and sudoku_v1 applications are from Heng Li’s “Programming Language Benchmarks” [57], and the remaining programs are from libraries: Michal Derkacz’s blas_d and blas_s [20], Dmitry Chestnykh’s password_hash and pbkdf2 [11], and Andre Moraes’ gocask [61]. The Name and LOC columns of the table give the name of the benchmark, and its size in terms of lines of code.
The inputs provided by the GCC suite for some of the programs are so small that they lead to execution times that, due to clock granularity, are too small to measure reliably. We gave some of these benchmarks larger inputs than the ones in the GCC suite. Where this was impossible or insufficient, we modified the program to repeat its work many times; the Repeat column shows how many.

The Alloc and Mem columns give respectively the number of objects allocated by each iteration of the program, and the amount of memory requested. These numbers were measured on the original version of each benchmark program, which used Go’s usual garbage collector. The Collections columns gives the number of collections in each iteration. (For the gocask benchmark, different runs of the program do different numbers of collections, due to the use of parallelism by a library.)

The last column group describes the results of our region analysis and its effects. The numbers come from a version of each benchmark program that was compiled to use our RBMM system. The Regions column gives the number of regions allocated during runtime of the program; the global region counts as one of these. The Alloc% column says what percentage of the allocations made by the program at runtime are from a non-global region, and therefore handled by our system. (The rest, the allocations from the global region, are handled by Go’s usual garbage collector.) The Mem% column says what percentage of the bytes allocated by the program at runtime are from a non-global region.

Table 3.2 contains our main performance data. Both column groups in this table compare the performance of each benchmark when compiled to use Go’s usual garbage collector (the columns labeled GC) and when compiled with our experimental RBMM system (the columns labeled RBMM, which also show the ratio between the GC and RBMM results). The column group named MaxRSS reports the maximum size, in megabytes, of the resident set of the program at termination, as reported by the GNU “time” command. Likewise, the column group named Time reports the wallclock (elapsed)
execution time of each benchmark in seconds.

We generated the two versions of each benchmark by compiling them with `gccgo` without any command line options beyond those selecting GC or RBMM, so all the programs were built at the default optimization level. To avoid measuring OS overheads, we disabled any output from the benchmarks during the benchmark runs. To eliminate the effects of any background loads, both the MaxRSS and Time results are averages from 30 trials.

The `gccgo` runtime system in Ubuntu’s libgo 0.4.6.1 provides a stop-the-world, mark-sweep, non-generational garbage collector. As usual, collections occur when the program runs out of heap at the current heap size. After each collection, the system multiplies the heap size by a constant factor, regardless of how much garbage has been collected.

We measured the base memory overhead of our RBMM system by compiling a Go program with an empty “main” function. This produces a binary of 13 KB; using GNU “Time” we find that the base MaxRSS value is 25 MB.

The `Collections` column in Table 3.1 shows the number of times Go’s garbage collector was woken up. The RBMM implementation only performs GC for data allocated from the global region, mostly by libraries that were compiled without RBMM. These counts were obtained from a single run of the benchmark, and do not reflect an arithmetic mean.

We used the numbers in the Alloc% and Mem% columns to cluster the benchmarks into three groups; the benchmarks in each group are sorted by name. For the programs in the first group, our system does virtually all memory allocations from the global region, handing responsibility for memory allocations back to Go’s garbage collector. For the programs in the second group, we do some allocations from non-global regions. For the programs in the third group, we do virtually all allocations from non-global regions, hardly using the garbage collector at all.

The benchmarks in the first two groups typically need more memory with RBMM than with GC, but the difference is small, and does not depend on how much memory the program allocates. This MaxRSS difference has
two sources. The first source is code size. The RBMM versions of the benchmarks have more code than the GC versions, for two reasons: first, the library that contains the implementation of all RBMM operations is included in the RBMM versions of benchmarks but not the GC versions, and second, the transformations of Section 3.7 only increase code size, and never decrease it. (The first effect is constant at 72 KB, while the second scales with the size of the program.) Since even a Go program that does nothing has a MaxRSS of 25.48 MB, due to the size of all the shared objects (such as libc) linked into every Go program, the benchmarks that report a MaxRSS around 26 or 27 MB in fact use about 1 or 2 MB of data. Therefore, for these programs, code size differences are a large part of the overall differences in MaxRSS (the maximum such difference is only 270 KB). The second source of difference in MaxRSS is that the RBMM versions need to allocate region pages, and since these programs do relatively few allocations using regions, not all the memory in these pages is used.

The MaxRSS results for the benchmarks in the third group show that if a program makes enough use of region allocations, the RBMM system can deliver an overall saving in memory usage. On all of these programs, the savings we achieve by freeing regions right after they become dead outweigh the extra costs of increased code size and additional internal fragmentation. For one of these benchmarks, binary-tree, the saving is significant. For the other three, the overall saving is more modest, but for meteor-contest and sudoku_v1, the saving in the part of the RSS we have control over, the part above the 25.48 MB RSS of the program that does nothing, the relative saving, is in fact quite significant.

With respect to timing, we get a big win on binary-tree, a program that was designed as a stress test for garbage collectors. It allocates many small nodes, which the GC system must scan repeatedly. The RBMM version can put all the nodes in regions where their memory can be reclaimed without any scanning. This makes the RBMM version more than five times as fast as the GC version, while using about 10% less memory.
Another version of this program, `binary-tree-freelist`, has its own built-in allocator, including a freelist; when a memory block is no longer needed, this version puts it into its own freelist, which is stored in a global variable. Later allocations get blocks from the freelist if possible. This ensures that all memory blocks ever allocated are not just reachable, but also potentially used throughout the program’s entire lifetime, which makes this a worst case for any automatic memory management system. Our region analysis detects that all this data is always reachable, so it puts all the data allocated by this benchmark into the global region, which is handled by Go’s garbage collector. So in this case the RBMM and GC versions actually do the same work and consume the same memory. However, the exact instruction sequences they execute do differ slightly, so their timing results differ too, probably due to cache effects. The results on this benchmark tell us that in this benchmarking setup, this speed difference of 1.6% is in the noise, and is not a meaningful difference.

We get a slightly higher speedup, 2.7%, for `gocask`. Since this program does allocate some memory from a non-global region, this speedup could conceivably result from region allocations, but since this program does very few of those, this speedup figure is also very likely to be noise. The same is true for all the deviations from 100% for all the other programs in the first two groups.

In the third group, one program, `binary-tree`, gets a spectacular, more-than-five-fold speedup, two have no change in speed, and the fourth program, `sudoku_v1`, gets a slowdown.

The original, GC version of `binary-tree` allocates a lot of relatively long-lived memory: it has the biggest MaxRSS of all our benchmarks. Each GC pass has to scan all this memory. The RBMM version of this program allocates all these nodes in regions, whose memory can be recovered without scanning their contents. Since the GC version spends most of its time in this scanning, avoiding these scans gives the RBMM version its huge speedup.

The next program in this group, `matmul_v1`, has very few allocations and
very few collections: apparently, most of the few blocks it allocates are very long lived. Because of this behavior, the GC version spends a negligible fraction of its runtime scanning the heap and freeing blocks, so the effect on the program’s overall runtime would also be negligible even if the RBMM version sped up this fraction of the program’s runtime by a large factor.

The **meteor-contest** program does about three and a half million allocations. In the RBMM version, each of these allocations has its own private region, so this version of the program does three and a half million region creations and removals. Hence, it recovers the memory of every block one by one, just like the GC version. The fact that we do not suffer a slowdown on this benchmark shows that our region creation and removal functions are efficient.

The **sudoku_v1** benchmark puts almost all of its memory in regions, and this allows it to use less memory than the GC version. Nevertheless, the RBMM version of this benchmark is slower than the GC version. We believe this happens because this benchmark has many function calls that involve regions, and the extra time spent by the RBMM version reflects the cost of the extra parameter passing required to pass around region variables.

### 3.17 Summary

In this chapter we have introduced a novel approach to fully-automatic memory management for the Go programming language employing region-based storage. It is based on a combination of static analysis to guide region creation, and lightweight runtime bookkeeping to help control reclamation.

Traditional region analysis algorithms propagate region information from callees to callers and vice versa. This means that any change to the program source code may require reanalysis of many parts of the program. If some of these reanalyses yield changed results, then these changes will have to be propagated through the program’s call graph. Reanalysis can end only when it reaches a fixed-point. In contrast, our analysis is context-
insensitive, allowing our system to propagate information only from callees to callers. This means that after a change to a function definition, we only need to reanalyze the functions in the call chain(s) leading down to it. We also introduced a novel concept of region protection counting, which acts as an efficient alternative to reference counting to prevent premature region removal. Chapter 5 extends the concepts presented in this chapter.
CHAPTER 4

Correctness of the RBMM Transformations

We adore chaos because we love to produce order.

M.C. Escher

This chapter presents a rigid argument for the correctness of the first-order subset of RBMM transformations explained in Chapter 3. To facilitate this argument we introduce a semantics that focuses on the relevant aspects of memory management while abstracting away the concepts that are less important for proving correctness.

4.1 Memory Management Correctness

Before a semantics is introduced, it is necessary precisely to define what is meant by having a memory management system that is “correct.” Much work on the correctness of memory management is for GC where correctness is defined in terms of what can be reached from the root set. As we have seen in Chapter 2 reachability gives a conservative approximation of memory safety. If we discuss the correctness of non-GC dynamic memory management then we need a model of memory operations that is more precise than that for GC but still simple. In this chapter we provide such a model in the form of a denotational definition. As pointed out by Morrisett et al., if we want to discuss memory safety beyond reachability then we need a model that precisely highlights the behavior of the memory operations [62].
For a program to uphold any reasonable degree of integrity, the safety of its memory management must hold. In the case of our RBMM transformations discussed in Chapter 3, the transformed program must behave “similarly” to the original program. This means that the transformed program should produce identical output as the unmodified program.

We are initially concerned with the question: “What does it mean to have a memory management system that is correct?” At the outset of things, we can broadly state that a correct system produces memory when requested, and recycles no-longer needed memory. However, that description is much too broad. For instance, it does not consider where the memory is produced from, or how the objects are associated to each other. We need a correctness statement that is sensitive to the relationships of objects that are managed by the memory system.

What we will show is that our system preserves the semantics of the original program, unmodified by our analysis and transformations. This assumes that the unmodified system (original compiler and runtime system) is correct. We are stating that idea of correctness by showing that our system preserves the behavior of the original program, even after our applied transformations. The only difference between programs generated by the original system and ours is how the memory is managed, which should have no impact on the semantics of the original unmodified program.

To demonstrate correctness we abstract away all aspects that are extraneous to proving this. We want to show that all objects that are bound to dynamic memory maintain the same shape and object-relations as the same objects produced by the original system. In other words, our transformations must preserve the points-to relationship between objects. This comes down to maintaining a certain equivalence between objects in the modified and unmodified memory systems, as will be explained below.

The following example illustrates the necessity for preserving both shape and value:
The function \texttt{shape\_and\_value} contains two fragments of code. In the first fragment of code, \texttt{bar} points to two separate objects that both have the same value. The second fragment shows \texttt{foo[0]} and \texttt{foo[1]} both pointing to the same object \texttt{a}, which has an integer member of value 42. The important concept is that while \texttt{foo} and \texttt{bar} contain similar values, their shapes
are completely different, as illustrated in Figure 4.1. foo has two pointers referring to the same object, while bar points to two separate objects. The transformations must preserve both shape and value for the program to behave as the original.

We abstract away memory addresses, since these are not important to show how our system maintains a certain bijection between objects of both systems. We also assume unlimited memory, as having a limit to memory is immaterial for this exercise. What allocator produces an object does not matter as long as the object is not used after it has been reclaimed. If we can demonstrate the bijection between objects created in both systems, and that our transformations do not alter the behavior of the program, then we can conclude that the behavior of our system is equivalent to that of the unmodified system. This bijection captures our idea of correctness in that the semantic equivalence between both systems holds. We also do not consider when objects are collected. RBMM and GC systems can reclaim objects at different times. Comparing when either system performs this reclamation is not necessary to prove correctness. The act of collecting an object is only in violation of correctness if that item will be accessed after it has been collected.

4.2 Semantic Language

To formalize our correctness argument we introduce a formal semantics that is sufficient to describe both our memory system and that of Go. The rules and items covered here will be explained in detail when we explain our transformation rules later. This semantics omit features of Go and GIMPLE (GCC’s intermediate representation which we use to build our RBMM system) that are not relevant to explaining our correctness argument. The transformations are source to source and represented by the function $T$.

The following disjoint sets are used to reason about specific types of data in our system. We use the $\$ character to represent the return value of
A grammar for the simplified language is given in Figure 4.2. The semantic domains are defined in Table 4.1. The selector is used to obtain a value located at a specified field within an aggregate data structure, or to retrieve a list item at the specified index. The selector abstracts away the notion of memory addresses, offsets of structure fields, and array indices. We assume the program is well-typed and that no array elements are accessed out of bounds. This abstraction permits our formalization to obtain the value from any scalar or aggregate data structure.
\[
\begin{align*}
\text{selector} &= \mathbb{N} \cup \text{ident} \\
\text{value} &= \text{prim} \cup \text{object} \cup \text{region} \cup \{\text{default}\} \\
\text{Environment} &= \text{ident} \rightarrow \text{value} \\
\text{Occupants} &= \text{region} \rightarrow (\mathcal{P}(\text{object}) \times \mathbb{N}) \\
\text{Store} &= \text{object} \rightarrow \text{selector} \rightarrow \text{value} \\
\text{RegionMap} &= \text{regionid} \rightarrow \text{region} \\
\text{State} &= (\text{Store} \times \text{Environment} \times \text{Occupants} \\
&\quad \times \mathcal{P}(\text{object} \cup \text{region}) \times \text{RegionMap})
\end{align*}
\]

Table 4.1: Semantic domains

A \textit{value} is our abstracted representation of items managed by our language, including instances of items that are allocated. \textit{default} represents the initial value of an item before any user code has had the chance to modify it. This represents a value of 0, or \textit{nil} in the case of a pointer value, as specified by Go semantics.

The definitions for the \textit{Store} and \textit{Environment} functions are similar. Our \textit{Store} models a program’s heap, or dynamic memory area. This area consists of memory that is not automatically allocated on the stack. The stack and register memories are modeled by the \textit{Environment}. To clarify, data can be stored in the \textit{Store}; however, the values of those data are only realized for computation when they are reachable from the \textit{Environment}. This distinction between two seemingly similar memory areas allows us to represent memory only relevant per function call (\textit{i.e.}, stack frames).

The \textit{RegionMap} introduces a level of indirection. A \textit{region} represents the contents of a region, and the \textit{regionid} represents a pointer to the \textit{region}. We call this pointer a \textit{region identifier}. \textit{RegionMap} is used to obtain the value pointed to by a given \textit{regionid}. Region identifiers are generated by the compiler during analysis and are dynamically allocated at runtime. This indirection is an important concept which will become apparent when region merging is discussed later in this chapter. All of the region operations take
region identifiers as arguments.

We use $\sigma$ to represent the Store, $\rho$ to represent the Environment, $\tau$ to hold an Occupants function, $\varphi$ to hold the set of free (unallocated) objects, and $\psi$ to hold the region identifiers. A region is a tuple, $(O, n)$, consisting of allocated values (occupants) and a protection counter value. The Occupants function is used to obtain the occupants, $O$, and protection counter value, $n$, from a specific region. A State is represented as $(\sigma, \rho, \tau, \varphi, \psi)$, where:

- $\sigma$: Store
- $\rho$: Environment
- $\tau$: Occupants function
- $\varphi$: Set of free objects or unused regions
- $\psi$: Region mapping

The semantic functions are of the following format, where the input to the semantic function is a statement and the current State. The result of evaluating the semantic function is an updated State. A $\ast$ character is the Kleene closure representing zero or more occurrences.

$$B :: Stmt^* \rightarrow State \rightarrow State$$
$$S :: Stmt \rightarrow State \rightarrow State$$

The semantic function $B$ for blocks is defined:

$$B[\;] = id$$
$$B[s; \tau] = B[\tau] \circ S[s]$$

The semantic function $S$ for statements is defined in Figure 4.3.

4.3 Transformation Correctness

We now show that the RBMM transformations, as presented in Chapter 3, preserve the semantics of the program being transformed. Of course, not all
\( S[x=y](\sigma, \rho, \tau, \varphi, \psi) = (\sigma, \rho[x \mapsto \rho y], \tau, \varphi, \psi) \)
\( S[x=y.s](\sigma, \rho, \tau, \varphi, \psi) = (\sigma, \rho[x \mapsto \sigma(\rho y)s], \tau, \varphi, \psi) \)
\( S[x.s=y](\sigma, \rho, \tau, \varphi, \psi) = (\sigma[o \mapsto (\sigma o)[s \mapsto \rho y]], \rho, \tau, \varphi, \psi) \) where \( o = \rho x \)
\( S[\text{return } y](\sigma, \rho, \tau, \varphi, \psi) = (\sigma, \rho[\$ \mapsto \rho y], \tau, \varphi, \psi) \)
\( S[v = f(\pi)](\sigma, \rho, \tau, \varphi, \psi) = (\sigma', \rho[v \mapsto \rho'(\$)], \tau', \varphi', \psi') \)

where \( \begin{cases} (\overline{p}, \overline{s}) = \text{lookup}(f) \\ (\sigma', \rho', \tau', \varphi', \psi') = B[\overline{s}](\sigma, \{p \mapsto \rho(\pi)\}, \tau, \varphi, \psi) \end{cases} \)
\( S[v = \text{AllocFromRegion}(r)](\sigma, \rho, \tau, \varphi, \psi) = \)
\( \begin{cases} \sigma[y \mapsto \lambda s.\text{default}], \rho[v \mapsto y] \\ \tau[\psi r \mapsto (O \cup \{y\}, n)], \varphi \backslash \{y\}, \psi \end{cases} \)

where \( \begin{cases} y = \text{select (object } \cap \varphi) \\ (O, n) = \tau(\psi r) \end{cases} \)
\( S[v = \text{new}()](\sigma, \rho, \tau, \varphi, \psi) = \)
\( \begin{cases} \sigma[y \mapsto \lambda s.\text{default}], \rho[v \mapsto y] \\ \tau[\psi G \mapsto (O \cup \{y\}, n)], \varphi \backslash \{y\}, \psi \end{cases} \)

where \( \begin{cases} y = \text{select (object } \cap \varphi) \\ (O, n) = \tau(\psi G) \end{cases} \)
\( S[r = \text{CreateRegion}()](\sigma, \rho, \tau, \varphi, \psi) = \)
\( \begin{cases} \sigma[r \mapsto x], \rho, \tau[y \mapsto (0, 0)], \varphi \backslash \{x, y\}, \psi[x \mapsto y] \end{cases} \)

where \( \begin{cases} y = \text{select (region } \cap \varphi) \\ x = \text{select (region}_{\text{id}} \cap \varphi) \end{cases} \)
\( S[\text{RemoveRegion}(r)](\sigma, \rho, \tau, \varphi, \psi) = \)
\( \text{if } n \neq 0 \text{ then } (\sigma, \rho, \tau, \varphi, \psi) \text{ else } \)
\( \begin{cases} \sigma[v \mapsto \bot | v \in O], \rho, \tau, O \cup \{r, \psi r\} \cup \varphi, \psi[r \mapsto \bot] \end{cases} \)

where \( (O, n) = \tau(\psi r) \)
\( S[\text{IncrProtection}(r)](\sigma, \rho, \tau, \varphi, \psi) = \)
\( (\sigma, \rho, \tau[\psi r \mapsto (O, n + 1)], \varphi, \psi) \)

where \( (O, n) = \tau(\psi r) \)
\( S[\text{DecrProtection}(r)](\sigma, \rho, \tau, \varphi, \psi) = \)
\( (\sigma, \rho, \tau[\psi r \mapsto (O, n - 1)], \varphi, \psi) \)

where \( (O, n) = \tau(\psi r) \)
\( S[\text{MergeRegion}(r_1, r_2)](\sigma, \rho, \tau, \varphi, \psi) = \)
\( \begin{cases} (O_1, n_1) = \tau(\psi r_1) \\ (O_2, n_2) = \tau(\psi r_2) \\ \hat{r} = (O_1 \cup O_2, n_1 + n_2) \end{cases} \)

Figure 4.3: The semantics of the memory-facing statements
aspects of the systems will need to be the same. For instance, the source of object allocations and free object sets will differ between the RBMM system and that of an unmodified system. In fact, we do not even model reclamation of the unmodified system, since all we are concerned with are allocated objects. What is important is that there exists a bijection between the sets of reachable objects and values of both systems.

We denote the domain of function \( f \) as \( \text{dom}(f) \). This represents the set of values \( x \) for which \( f(x) \) is defined.

With \( \text{dom} \) defined we now provide the following two definitions and an invariant that holds across all of the semantic functions defined above. This invariant must hold for our system to be correct.

Let \( o \) be an object and \( \sigma \) be a \emph{Store}. We define the set of objects reachable from \( o \) as:

**Definition 1.**

\[
\text{Reachable}(\sigma, o) = \{o\} \cup \bigcup_{s \in \text{dom}(\sigma o)} \text{Reachable}(\sigma, \sigma os)
\]

Let \((\sigma, \rho, \tau, \varphi, \psi)\) be a runtime state. We can now define the set of objects reachable from the root set as:

**Definition 2.**

\[
\text{RR}(\sigma, \rho, \tau, \varphi, \psi) = \text{object} \cap \bigcup_{i \in \text{dom}(\rho)} \text{Reachable}(\sigma, \rho i)
\]

**Invariant 1.** The system will never reclaim a region if any of the objects belonging to it are can be accessed at a later program point. Formally: Let \((\sigma, \rho, \tau, \varphi, \psi)\) be a runtime state on arriving at a \texttt{RemoveRegion}(r) instruction, then \( \psi r \notin \varphi \land (\tau(\psi r) = (O, 0)) \Rightarrow O \cap \text{RR}(\sigma, \rho, \tau, \varphi, \psi) = \emptyset. \)

In other words, when a \texttt{RemoveRegion} operation is executed, and the region can be successfully reclaimed (a protection count of zero), then none of the objects in the region are reachable from the root set.
To define semantic equivalence between two states we first begin with a convenience function, \( e \), which captures pairs of corresponding objects and values from both states. We decorate items from the transformed system by using a prime.

\[
e(\sigma, v, \sigma', v') = \{(v, v')\} \cup \bigcup_{s \in \text{dom}(\sigma v) \cup \text{dom}(\sigma' v')} e(\sigma, \sigma vs, \sigma', \sigma' v' s)
\]

This routine recursively traces an object \( v \) in the Store \( \sigma \) by following the fields (selectors), \( s \) within \( v \). The result is a set of pairs of related values from both systems \( \sigma \) and \( \sigma' \).

The following predicate aids the definition of semantic equivalence:

\[
bij(v, v', p) \iff \forall (w, w') \in p. (v = w \iff v' = w')
\]

where \( p = e(\sigma, v, \sigma', v') \)

This predicate takes a pair of objects \( v \) and \( v' \) and a set of pairs of values \( p \) from \( e \). The predicate \( bij \) is only True if \( v \) and \( v' \) are paired with no other objects but each other from the specified pairs.

We can now formally define semantic equivalence between the values \( v \) and \( v' \) and object associations between stores \( \sigma \) and \( \sigma' \):

\[
(\sigma, v) \equiv (\sigma', v') \iff \forall (x, x') \in p. \begin{cases} bij(x, x', p) & \text{if } \{x, x'\} \subset \text{object} \\ x = x' & \text{otherwise} \end{cases}
\]

where \( p = e(\sigma, v, \sigma', v') \)

This statement demonstrates an equivalence between object shapes and values via a recursive traversal of each pair of objects. The predicate \( bij \) only guarantees that the shape of the two tested objects are the same. The equivalence statement above ensures that both the object shapes are identical as well as their values.
The states of both systems are equivalent if the values of all the variables in the environment are equivalent:

\[(\sigma, \rho, \tau, \varphi, \psi) \equiv (\sigma', \rho', \tau', \varphi', \psi') \iff \forall i \in \text{dom}(\rho). (\sigma, \rho_i) \equiv (\sigma', \rho'_i)\]

Given this definition of state equivalence, we can now state the following theorem of semantic equivalence. It states that the transformations applied by our RBMM system do not alter the semantics of the original program.

**Theorem 1.** For all function bodies \(B\), and states \(s\), \(B[s] \equiv B'[s]\), where \(B' = T(B)\) and \(T : \text{Stmt} \rightarrow \text{Stmt}^*\) is the RBMM transformation function.

### 4.3.1 Region Creation and Removal

Any program not transformed by our system holds trivially to Theorem 1, since no code transformations would have been performed to the analyzed program. We now present a few simple cases to demonstrate the correctness of our RBMM transformations in terms of Theorem 1.

Recall that all of the transformations are performed automatically at compile-time, and require no annotations by the programmer. The first series of transformations we describe below are the region creation and removal operations. These are the most important transformations since they are the routines that produce and remove regions at runtime. No other region operation can occur without first having a region to manipulate.

When the creation and removal operations are added to the identity transform, \(T\), Theorem 1 holds trivially, since the regions will never be used and the program semantics will remain unchanged. Additionally, if a region is removed that is never used, the semantics of the program will also remain unchanged.

We introduce three invariants that pertain to the creation and removal operations:
1. **CreateRegion** maintains the invariant that a region will always be created for an object before that object is first used. This ensures that an allocation for that object will be produced from a region that exists.

2. **RemoveRegion** maintains the invariant that a region will always be removed if its objects are no longer used in the program or do not escape the function. If a removal is called on a region and the objects are referenced later, then **RemoveRegion** will return early and not be reclaimed.

3. All functions will try to reclaim the regions for each of the variables that are passed to it and for any variables that are allocated within the function that might not be accessed later. Objects that are returned from a function are created from a region that is passed to that function by the caller. It is safe for the callee to try to remove that region, since the caller would have incremented the region’s protection counter if the caller needs the returned object later.

```plaintext
func f(f_1, ..., f_n) { 
    S_1; ... S_m;
    return f_0;
} 
```

The above transformation is the simplest implementation of region creation and removal operations. In this case, a traditional function is transformed by adding region creation operations to the function prologue and any removals are added in the function’s epilogue. By placing the creation operations in the function’s prologue we guarantee that the region will be available for the
function to use, since all uses of regions occur after the function prologue. Additionally, by placing the removal operations in the function epilogue we guarantee that the region will not be prematurely reclaimed. Again, if a region is removed that is never used, the semantics of the program is not altered. In this case, since all regions are created before any statements of the original function are executed, we guarantee with semantic rule CreateRegion (copied below) that all region creations will produce a region before that region is needed.

\[
S[r = \text{CreateRegion}()]((\sigma, \rho, \tau, \varphi, \psi) = \\
(\sigma[r \mapsto x], \rho, \tau[y \mapsto (\emptyset, 0)], \varphi \setminus \{x, y\}, \psi[x \mapsto y])
\]

where \[
\begin{align*}
y &= \text{select} (\text{region} \cap \varphi) \\
x &= \text{select} (\text{region}_\text{id} \cap \varphi)
\end{align*}
\]

All regions will be available before they are needed, and a program will not try to access a region that does not exist. CreateRegion defines \(y\) as being obtained via select from the free regions in \(\varphi\). \(y\) is initialized to having an empty occupants set \(O\), and a protection counter starting at 0. \(\psi\) establishes the mapping of the \(\text{region}_\text{id} \ x\) to the contents of \(y\). The Environment remains unchanged and the Store maps the local region variable \(r\) to the value \(y\). Both \(y\) and the identifier \(x\) (which points to \(y\)) are produced from the set of free objects \(\varphi\). Only a region is produced, which objects can be allocated from.

It is also safe to place a RemoveRegion operation at the end of the function. Recall the semantic rule for RemoveRegion:
When a `RemoveRegion` operation is encountered two things can happen: either the protection counter is non-zero and the region will not be reclaimed, or the protection counter is zero and the region’s memory will be reclaimed. This rule states that if the protection counter $n$ is zero then the occupants of the region are set to $\bot$ and their memory is returned back to the free store $\varphi$. Similarly, identifier $r$ for the region is mapped to $\bot$ and is also returned to the free store $\varphi$. Our notion of protection counters places the responsibility of region removal in the caller routine. If a function call is made, our analysis will first check if any variables belonging to any regions passed to the callee are used later. If this is the case, then the protection counter is incremented for the region. This non-zero protection counter will ensure that the callee cannot remove the region for a variable that might be used later in the caller.

If the protection counter is zero, any subsequent `RemoveRegion` call will return the region back to the set of free/unused objects $\varphi$. This rule preserves Invariant 1 since it does not reclaim any memory. However, because the protection counter is zero, a following reclamation operation can successfully reclaim the region. Since our system only reclaims memory at the latest program point in the function where all items are no longer accessed, we can guarantee that Invariant 1 is not violated. Additionally, if a region is passed to a function, and that region contains objects that will outlive the function call, then the protection counter will be non-zero, and thus the callee will not successfully reclaim the region if it were to call `RemoveRegion`.

The `CreateRegion` transformation does not modify the semantics of the program. This transformation does not modify any variables of the original
program, and the only objects manipulated are those created on behalf of the RBMM system. We claim this transformation maintains equivalent semantics to that of the unmodified system and is correct since the bijection is not violated. This reasoning satisfies Theorem 1.

Building on the previous transformation, we now introduce the following where region creation is moved from the prologue into a later point of the function:

\[
\begin{align*}
S_1; & \ldots S_m; \\
S_{m+1}; & \ldots S_n; \\
\Rightarrow & \\
r = \text{CreateRegion}(); \\
S_1; & \ldots S_m; \\
S_{m+1}; & \ldots S_n;
\end{align*}
\]

where \( r \notin \text{used}(S_1; \ldots S_m) \)

It is safe to place a creation operation before the first use of that region. Logically, this case is safe, because the transformation can move a creation before the first statement that will require the region. From the perspective of a Go program, this is the program point where the programmer issues a call to Go’s new or make operations. new or make are replaced by our AllocFromRegion operation, therefore a region must be available to allocate from at that program point. Placing a region creation before the first use of the region ensures that the region will be available for allocation. This transformation preserves Theorem 1 because the program semantics remain unchanged.

We now introduce another transformation for handling region removal:

\[
\begin{align*}
S_1; & \ldots S_m; \\
S_{m+1}; & \ldots S_n; \\
\Rightarrow & \\
\text{RemoveRegion}(r); \\
S_1; & \ldots S_m; \\
S_{m+1}; & \ldots S_n;
\end{align*}
\]

where \( r \notin \text{used}(S_{m+1}; \ldots S_n) \)
A region reclaim operation can be moved to a point in a function when all occupants allocated from the region are no longer needed, in other words a program point where none of the region's occupants will be accessed anymore. Our static analysis keeps track of which occupants are allocated from which regions, and which program points each occupant is referenced by. The reclaim operation is inserted at the earliest program point where the region's occupants will no longer be referenced. The protection counter (discussed later) will prevent a region from being removed prematurely (e.g., if any of its occupants are still needed). In fact, it is safe to place multiple reclaim operations of the same region throughout the function. This transformation holds for Theorem 1 since the program semantics will remain unchanged. The region removal will only occur when its occupants are no longer used.

### 4.3.2 Function Definition and Application

The next set of transformations guide the conversion of a traditional Go function prototype into one which contains formal parameters that accepts region arguments.

\[
\text{func } f(f_1, \ldots, f_n) \{ \\
\quad S_1; \ldots S_m; \\
\quad \text{return } f_0; \\
\} \quad \Rightarrow \\
\text{func } f(f_1, \ldots, f_n)(r_0, r_1, \ldots, r_p) \{ \\
\quad S_1; \ldots S_m; \\
\quad \text{return } f_0; \\
\}
\]

where \( \langle r_0, \ldots, r_p \rangle = \text{ir}(f) \)

The above transformation does not modify any objects. The state of the program will remain unmodified, therefore this transformation is immediately correct and holds to Theorem 1.

The next transformation represents function application when regions can be passed as arguments and/or returned.
\[ v = f(v_1, \ldots, v_n) \mapsto v = f(v_1, \ldots, v_n)\langle r_0, r_1, \ldots, r_p \rangle \]

where \( \langle r_0, \ldots, r_p \rangle = \text{compress}_f(R(v_0), R(v_1), \ldots, R(v_n)) \)

The above transformation is guided by the semantic rule for function application:

\[
S[v = f(\overline{a})](\sigma, \rho, \tau, \varphi, \psi) = (\sigma', \rho'[v \mapsto \rho'(\$)], \tau', \varphi', \psi')
\]

where \[
\left\{ \begin{align*}
(p, s) &= \text{lookup}(f) \\
(\sigma', \rho', \tau', \varphi', \psi') &= B[\overline{s}](\sigma, \{p \mapsto \rho(\overline{a})\}, \tau, \varphi, \psi)
\end{align*} \right.
\]

The \text{lookup} function produces the parameters and statements for a function \( f \). The tuple produced by \text{lookup} is used to generate the \text{State} for \( f \) by mapping the \text{Environment} for each parameter, \( p \), of \( f \). The second statement in the \textbf{where} clause of this rule defines \( f \)'s state as being the resulting state after executing all statements in \( f \). Parameters \( p \) passed to \( f \) are mapped to the input arguments \( a \).

The regions described in the first transformation in this section, where the function definition is transformed, are now passed to the function application in the second transformation. The same regions presented in the caller are those passed down to the callee. Note that this transformation does not allocate from a region and that no reclaim has occurred.

Invariant 1 is preserved by this transformation. If our analysis determines that any objects belonging to any of the region arguments passed as input to the callee are needed later, then an \text{IncrProtection} operation is inserted just before the call site, and a \text{DecrProtection} operation is inserted immediately following the call site. This guarantees that Invariant 1 will hold.

Function application is correct in our system and holds for Theorem 1 since it maintains the same state semantics as that of the unmodified system.
The only difference is the addition of region parameters, which act just as any other parameters and have no immediate effect on the state. Since no memory allocations or reclamations occur by the mere act of function application the transformation preserves correctness.

4.3.3 Region Protection Counting

The following transformation describes where our analysis places the increment and decrement protection counter operations. \texttt{IncrProtection} is placed before and \texttt{DecrProtection} is placed after a call site where any objects belonging to a region can be accessed later in the caller:

\[
\begin{align*}
S_1; \ldots S_m; \\
v = f(\ldots)(\ldots,r,\ldots) \\
S_{m+1}; \ldots S_n; \\
\end{align*}
\]

\[
\begin{align*}
\leadsto \quad &\begin{array}{c}
S_1; \ldots S_m; \\
\text{IncrProtection}(r); \\
v = f(\ldots)(\ldots,r,\ldots) \\
\text{DecrProtection}(r); \\
S_{m+1}; \ldots S_n;
\end{array} \\
&\text{where } r \in \text{used}(S_{m+1}; \ldots S_n)
\end{align*}
\]

The relevant semantic rules are:

\[
S[\text{IncrProtection}(r)](\sigma, \rho, \tau, \varphi, \psi) = (\sigma, \rho, \tau[\psi r \mapsto (O, n + 1)], \varphi, \psi)
\]

\[
\text{where } (O, n) = \tau(\psi r)
\]

\[
S[\text{DecrProtection}(r)](\sigma, \rho, \tau, \varphi, \psi) = (\sigma, \rho, \tau[\psi r \mapsto (O, n - 1)], \varphi, \psi)
\]

\[
\text{where } (O, n) = \tau(\psi r)
\]

After this transformation a region will have a protection counter value greater than one if any of its occupants can be accessed later.

This transformation preserves Invariant 1. No user objects are modified and no objects are allocated or reclaimed by incrementing or decrementing a protection counter. Additionally, since the semantics of the program is

108
unchanged, Theorem 1 holds.

4.3.4 Region-Based Allocation

We now prove the correctness of region allocation by showing the equivalence between program states between the original program and the transformed program. This is the most interesting of the transformations described because it associates objects allocated in the user’s program with regions. The following transformation shows the result of our compile-time analysis transforming the original allocation code into our RBMM equivalent. In the following case, our static analysis locates a new statement in Go and transforms it into the following representation:

\[
\begin{align*}
  v & := \text{new} \ (t) \\
  \leadsto \\
  v & := \text{AllocFromRegion} \left( R(v) \right)
\end{align*}
\]

This transformation is responsible for producing a piece of freely available dynamic memory from region \( r \) and binding the result to the variable \( v \). This transformation moves all allocations in the program to regions. This means that an object cannot be removed until the region it was produced from has a protection counter value of zero.

The semantic rule for \text{AllocFromRegion} shows what happens to the program’s state when a program calls the allocation routine \text{new} in Go. The \text{Store}, which contains the memory that \( v \) is bound to, is initially mapped to the result of executing an abstract function that initializes the memory assigned to \( y \) to a default value. \( y \) is the \text{Store}’s actual memory that will be bound to \( v \). The \text{where} clause of this rule defines \( y \) as being an object produced from \( \varphi \) (the set of free objects). The \text{Environment} is updated such that \( v \) maps to the \text{Store}, specifically \( y \) in the \text{Store}. The \text{Occupants} function shows that \( y \) belongs to the set of objects \( O \) in \( r \), and that the protection count \( n \) for \( r \) is not modified.

This transformation rule guarantees that a value will be produced by \text{AllocFromRegion}. The allocated memory is produced from free memory
that is associated to the Store and managed by a region \( r \). Since our abstract system is unbounded in memory, the case of running out does not occur. Memory produced from the free area is removed from the free area which prevents live memory from being accidentally (and incorrectly) allocated twice. Invariant 1 holds after this transformation, since the transformed program only produces memory from \( \varphi \) and does not perform any reclaim.

The bijection holds for this transformation, which can be seen in the semantically equivalent states of \texttt{new} and \texttt{AllocFromRegion}. This also means that Theorem 1 holds. These two rules are copied below:

\[
S[v = \texttt{AllocFromRegion}(r)](\sigma, \rho, \tau, \varphi, \psi) = \\
\left( \sigma[y \mapsto \lambda s. \texttt{default}], \rho[v \mapsto y], \tau[\psi r \mapsto (O \cup \{y\}, n)], \varphi \setminus \{y\}, \psi \right) \quad \text{where} \\
\left\{ \begin{array}{l}
y = \text{select } (\text{object } \cap \varphi) \\
(O, n) = \tau(\psi r)
\end{array} \right.
\]

\[
S[v = \texttt{new()}](\sigma, \rho, \tau, \varphi, \psi) = \\
\left( \sigma[y \mapsto \lambda s. \texttt{default}], \rho[v \mapsto y], \tau[\psi G \mapsto (O \cup \{y\}, n)], \varphi \setminus \{y\}, \psi \right) \quad \text{where} \\
\left\{ \begin{array}{l}
y = \text{select } (\text{object } \cap \varphi) \\
(O, n) = \tau(\psi G)
\end{array} \right.
\]

The only difference between these rules is where the allocated memory \( y \) comes from. This is why there are different \( \tau \) statements between the systems. In the case of Go’s system, the memory \( y \) is produced from \( G \), whereas in our RBMM system the memory is produced by the specified region \( r \). From the perspective of an abstracted system, both \( r \) or \( G \) are sources for memory, and thus the details of where allocated memory comes from does not affect the semantic equivalence since the values themselves are not modified. The result is the same, both functions return a piece of memory from the Store and bind it to a variable, \( v \) or \( v' \) in the Environment.

Our system protects the contents of a region via our concept of protection counters. In the case that an object allocated from a region is reached later, and that region is passed to a callee, the callee must somehow be told that it cannot reclaim that region. The purpose of a region’s protection counter
is to tell the callee that a region is needed later by a caller higher in the call-chain, and therefore the callee cannot reclaim that region. We insert \texttt{IncrProtection} and \texttt{DecrProtection} in pairs. The former is placed before a function call and the latter just after. This ensures that if an object from a region is needed after the call then its counter will be non-zero when entering the callee. Hence, the callee is unable to reclaim the region. After the callee completes, the counter is decremented and thus will have the same value it had before executing the callee.

If a region is needed for an object, then that region is passed with the objects to the callee. That is just what our function application transformation accomplishes. Thus, the allocated object will be alongside its region and cannot be removed by the callee unless the region’s protection counter is zero. Therefore, the object will not be reclaimed if it or any of the other objects from its region are accessed after the callee.

Our unification analysis will associate any objects that can be accessed from any other objects into the same region. Therefore, the region for any object referred to directly or indirectly cannot be reclaimed if any of the region’s occupants can be reached later. This unification ensures that when a region can be reclaimed, the objects belonging to the region are no longer needed.

This transformation preserves Invariant 1 since our system will not remove an allocated object if it is needed later. Theorem 1 holds because the semantics of the program are unchanged. All allocations will remain for the program to use as long as that allocation can be accessed later. The only way the allocation will be removed is if the protection counter for its region is zero; meaning that the region’s occupants will not be accessed later.

\subsection*{4.3.5 Region Merging}

Region merging is an optimization we have implemented in our original proof-of-concept described in Chapter 3. There are cases, as mentioned in Section 3.13, whereby two regions become associated at runtime and must
be considered the same.

The relevant semantic rules are:

\[
S[[\text{MergeRegion}(r_1, r_2)]](\sigma, \rho, \tau, \varphi, \psi) =
\]

\[
(\sigma, \rho, \tau[\psi r_1 \mapsto \hat{r}], \varphi[\psi r_2 \mapsto \psi r_1]) \quad \text{where} \quad
\]

\[
\begin{cases}
(O_1, n_1) = \tau(\psi r_1) \\
(O_2, n_2) = \tau(\psi r_2) \\
\hat{r} = (O_1 \cup O_2, n_1 + n_2)
\end{cases}
\]

The \text{MergeRegion} operation is responsible for taking two region identifiers and performing a union on both of their contents. In this transformation the contents of \(r_1\) remains and the contents of \(r_2\) is copied (unioned) with that of \(r_1\). \(\hat{r}\) represents the union of both regions. Once \(r_2\) has been copied, its contents is reclaimed, and \(r_2\) is updated to point to the same data as \(r_1\), which is \(\hat{r}\). This rule is why we have to make the distinction between a region header and region contents. Because both \(r_1\) and \(r_2\) point to the same region, any subsequent region operation on either will result in the manipulation of \(\hat{r}\), which they both refer to.

The protection counting scheme is correct. For every increment operation, \text{IncrProtection}, there will be a matching decrement operation, \text{DecrProtection}. Recall that the counters are incremented before a function call, \(f\), if a region is needed after that call. If a counter is incremented, then it will have a matching decrement immediately following the call to \(f\). In the case of a merged region, since both of the identifiers \(r_1\) and \(r_2\) still remain, they will also visit all of the decrement operations that were to be encountered. Since these regions now refer to \(\hat{r}\), which has the summed protection counter for both \(r_1\) and \(r_2\), our scheme of matching increment and decrement operations will still hold, and thus \(\hat{r}\) will be removed when all of the \(O\) from \(\hat{r}\) are no longer reachable. Additionally, \text{MergeRegion} preserves our correctness statements, as no regions are being allocated or removed from. The regions which have been merged will still adhere to the semantics of the previous transformations, which we have shown to hold for
Theorem 1.

4.4 Conclusion

We have shown that the memory facing part of the language affected by the RBMM transformations preserve Invariant 1. None of the transformations change the values and shapes of the program’s objects. Because Invariant 1 is preserved, no reachable objects are reclaimed. The bijection shows that the values and shapes of objects are preserved. Therefore, the semantics of both programs under the different memory systems are equivalent. This shows that our system holds with respect to Theorem 1.
5.1 Introduction

RBMM is not without its limitations. In Chapter 2 we discussed the region bloat problem, whereby a region might contain numerous objects with a majority of them being dead (never used again). In such a case, the memory footprint of the process is unnecessarily larger than it really has to be. Since a region cannot be reclaimed until the program point where none of its objects are referenced again, a region might be kept alive just from a single object that is referenced at a later program point. In this chapter we introduce several enhancements to the RBMM system described in Chapter 3. First, we modify our unification algorithm to allow a single object to belong to multiple regions, if such an object contains pointers to other data types. The latter enhancement opens up the possibility of smaller regions that do not have to contain the encapsulating data structure. Secondly, we try to improve a process’s memory footprint by solving the memory bloat and global region problem. For the latter problem, we introduce a garbage collector, which is capable of reclaiming dead objects from regions. This combination of RBMM and GC tries to achieve the advantages of both systems while avoiding the disadvantages. Along the way we present three additional contributions:
• We propose a new way of combining GC and RBMM, with less overhead than similar systems [38]. Our design treats each region as being divided into a to and from semi-space that uses the Big Bag of Pages [71] concept for managing type information of allocated objects.

• Our algorithms support partial collections: recovering memory from some regions, but not all.

• We enable the collection of segments of arrays. In languages that support slices, it can happen that some elements of an array are reachable while others are not. We show how to recover the memory occupied by the dead elements, at a low cost.

Before we discuss our design of a region-aware GC we first mention some of the changes to our RBMM system since Chapter 3. These changes are discussed in the next section, with our RBMM and GC solution design immediately following that in Section 5.3.

5.2 Enhancing the Analysis

Our RBMM system utilizes a unification method that places all items into the same region if there is a points-to association between them. This means that any item that contains a pointer as a field will also share the same region with the field. While this field-insensitive unification is simple to implement, it can be made more precise. Recall that we want to free items as soon as possible to produce a smaller memory footprint. In the case of a structure that can contain pointers to other items, we want to allocate the data for its pointer objects from separate regions. This produces a situation where a structure can be reclaimed in pieces. For example, consider a linked-list. If we can separate its skeleton from the data, then the skeleton can be reclaimed separately from its data.

To accomplish this enhancement we modify two analysis rules from Chapter 3. We repeat those rules in Figure 5.1. Our modified rules are seen in
This rule change is rather simple; however, the cost of implementing this modification has been high (it took a long time). For instance, functions no longer require a single region per argument, instead, they now require multiple arguments per region (including any regions that might be needed for the function’s return value).

### 5.2.1 Increasing Conservatism for Higher-Order Functions

In our original implementation discussed in Section 3.9 we inserted trampoline functions that would dynamically merge two regions at runtime if any of the variables between regions became associated to each other. We noticed that runtime merging was harming the system’s performance. Additionally complicating matters is the fact that a non-global region might get merged with the global region. This is dangerous, as it means that all data from the non-global region can never be reclaimed, not even by Go’s existing garbage collector. This occurs because the memory has been allocated from our RBMM allocator and then gets merged and becomes managed by the global region. Since the global region is managed by Go’s existing garbage collector, and it never allocated these addresses, it cannot reclaim them. Therefore, runtime merging can effectively introduce memory leaks into a program.

To eliminate these problems we relaxed our semantics such that all variables that are passed as arguments to closures belong to the global region. Figure 5.3 describes this change in terms of our semantics. This rule applies only to closures. This is overly conservative but means that Go’s garbage

\[
S[v_1 = v_2, s] \rho = (R(v_1) = R(v_2))
\]

\[
S[v_1.s = v_2] \rho = (R(v_1) = R(v_2))
\]

Figure 5.1: Original constraint rules
\[ S[v_1 = v_2.s] \rho = (R(v_1) = R(v_2)) \]
\[ S[v_1.s = v_2] \rho = (R(v_1.s) = R(v_2)) \]

Figure 5.2: Modified constraint rules

\[ S[v_0 = f(v_1 \ldots v_n)] \rho = \bigwedge_{i=0}^{n}(R(v_i) = \mathcal{G}) \]

Figure 5.3: Added constraint rule for closures

collector can be used to allocate and manage their memory (reducing memory leaks). It also means that we eliminate the overhead of runtime merging, effectively replacing such overhead with that of a garbage collector.

In this chapter we introduce a region-aware garbage collector, so as to enable collection of data from the global region, and avoiding the reliance on Go’s existing collector for such matters.

5.3 Combining Regions and Garbage Collection

Recall that automated memory management systems can be implemented using either RBMM or GC. Since an RBMM system does not require a scan of the program’s memory at runtime, it can result in a faster running executable than a GC system. However, RBMM suffers from the region bloat problem as discussed in Chapter 2. In other words, there is a time-space trade-off between the systems.

However, the situation is more complicated than that. Ideal RBMM should not only produce smaller execution times, but should be able to realize a potential to use less memory than GC. This potential is due to two facts:

- RBMM needs considerably less memory for its own bookkeeping, and

- RBMM can decide what memory to free based on what the program will need in the future, rather than on what it can currently access.
In principle, RBMM should be able to reclaim memory in relatively small chunks, resulting in a flatter memory footprint. However, there will always be programs exhibiting behaviors that favor GC.

As discussed in Section 2.3.5, object lifetime is an undecidable property of a program [54, 66]. Our analysis relies on program points for determining if an allocated object will be used later. Since allocations are associated to pointer objects, and the lifetime of a memory item is in general undecidable, region analysis must conservatively approximate the item’s lifetime. In some cases, the approximation is too conservative, creating long-lived regions in which many items are no longer needed. In particular, items referred to by global variables are placed in a region which will be kept until the program exits. Several previous studies [37, 7] have shown that such long-lived regions can accumulate large numbers of now-dead objects beside some live ones, increasing the program’s memory footprint significantly, and in some cases beyond the limit of acceptability.

In Chapter 3 we treated the region containing global variables specially by managing it with Go’s existing built-in GC system, but this approach has two shortcomings. First, it does nothing to reclaim unused items in other long-lived regions, and second, it leaves us with two non-interoperable memory management systems. The reason why the second point matters is that it prevents us from implementing an optimization that is potentially important. Recall the region merging discussion in Section 3.13. In certain cases, one code path may permit two regions to be kept separate, while a less common code path may require the analysis to consider them to be the same region. Keeping them separate may permit one region to be reclaimed much earlier than the other, so we would prefer to do this. However, we cannot do this unless we can merge the two regions at runtime. Therefore, we cannot merge regions when a subset of the allocations are dependent on Go’s existing garbage collector.

In this chapter we modify our RBMM system to allow the contents of regions (especially long-lived regions) to be garbage collected. We still expect
that most regions will be short lived, and that most items will be recovered without GC, when the regions containing them are removed. Since we expect GC operations to be the exception and not the rule, we want region operations (the creation and destruction of regions, and allocation of memory from regions) to be as fast as they are in our RBMM system presented in Chapter 3. The existence of the GC system should not have a significant impact on the performance characteristics of regions that are never garbage collected. A secondary goal is to make our GC system a moving collector (see Section 2.3.2), to improve locality of reference and to reduce internal fragmentation.

5.4 Managing Ordinary Structures

This section describes changes to our RBMM design to facilitate the introduction of a region-aware garbage collector.

We want our garbage collector to be a moving collector. Such a collector needs to know which parts of each item are pointers and which are not. The
simplest way to give it this information is to include type information next to every item in every region [38]. Adding a type description next to every item in a region would significantly increase memory consumption. Since each region will contain values from only a limited set of types, we can greatly reduce the space overhead of type information by storing the description of each type that can occur in the region just once, and associating all values of that type in the region with that description. In other words, we split each region into a set of zones, with one zone per type that can appear in the region.

If there are $N$ types that can appear in a region, then this scheme costs us the memory occupied by $N$ zone headers, each of which is 72 bytes in size (the zone header’s fields will be discussed later). Typical values of $N$ (from our test cases) range from 1 to 25, so the typical overhead ranges from 72 to 1,800 bytes per region. This is a fixed cost. On the other hand, the memory savings that this scheme allows scales with the amount of data in the region. For example, if a region contains 100,000 8 byte items (800 KB total) and 200,000 24 byte items (4.8 MB total), then, by not having to identify the type of each item with an 8 byte pointer to type information, we save $300,000 \times 8 = 2.4$ MB, which in this case represents 50% overhead. In other cases, the percentage will be different. However, it should be clear that our design saves not just significant amounts of memory, but also the time needed to fill in this memory.

Region inference can give us, for each of the regions it creates, the set of types whose values may appear in the region. In fact, it is guaranteed to do so, unless the program uses language constructs (such as interfaces) that introduce polymorphism and thus hide the actual types of some values from the compiler. Section 5.6 discusses how to deal with interfaces. Until that section is reached we will assume interfaces are absent, and that we do know the set of types in each region.

Figure 5.4 shows the effect of splitting a region into a set of zones, one for each type in the region, each zone holding all the items of that type in
the region. Each zone consists of a list of flexipages, and has a header that contains the following slots:

- A pointer to the header for the whole region.

- A pointer to the next free byte of data on a page within the zone which can be used to fulfill the next allocation request. This is a small optimization eliminating the need for each page to have a pointer to its next free byte.

- A pointer to a description of the type of the items in the zone, which we call a typeinfo. These typeinfos are read-only data structures created by the compiler. Each typeinfo contains the size of the type, if the type is a pointer, and a flag letting us know if the type is a special built-in (e.g., slice). Three other values in the typeinfo are useful for managing structures; these additional fields let us know the number of fields, the offset of the field, and an index in the typeinfo table referring to the next field in the structure.

- The number of items of this type that fit in a single page, i.e. in a flexipage of the minimum size. This is calculated from the page size, the size of flexipage headers, and the size of each item. We will show the exact formula later.

- A pointer to the start of the most recently allocated flexipage in the zone. Note that we also maintain a pointer to the last page in the zone’s flexipage list. This allows us to quickly return this list to the global freelist of pages when we reclaim the zone. To aid cache locality, we treat a zone’s flexipage list like a stack. The most recently used page is at the top of the stack, and after zone reclamation the freelist will have that most recently used page at the top of its stack. Upon the next request for a free page, the top of the freelist will be returned which happens to be the most recently used freed page.
• During a collection, the pointer above defines the list of flexipages that act as the from-space. We also have a corresponding pointer that serves to define the to-space. This second pointer is used only during collections. As with the former, we also maintain a pointer to the last page in the to-space for the reasons discussed in the previous bullet.

• A flag telling us if the zone is to contain a special data structure that we garbage collect differently from all other types (e.g., slices). Note that this field is also in the typeinfo and we should be able to optimize it away.

Therefore, a zone in this modified approach acts similar to what a region did in Chapter 3, with the exception that all items in a zone are of the same type. The region now becomes just a container of zones, and the memory that is distributed via region allocation will come from a flexipage located in the appropriate zone of the region.

The region header contains:

• The number of zones in the whole region. This allows our runtime system to manage a region and all of its zones properly. For instance, when a region is deleted, our runtime system must know how many zones a region has so that each zone in a region’s array of zone headers can be visited and removed.

• The region’s protection counter, which prevents premature region removal.

• Two bits that are needed only during collections. These bits permit our GC algorithm to selectively collect from a subset of all regions within the program. The REGIONBEINGCOLLECTED bit is set iff the collection is attempting to recover memory from the region, while the REGIONNEEDSTRACING bit is set iff the GC algorithm needs to traverse the contents of the region in order to find reachable items in the regions being collected. Note that our implementation experimented
with in this chapter uses just a single boolean flag to determine if a region has/is being garbage collected. When we build our system to not use our GC, this flag is not in the resulting build.

- An array of the headers of the region’s zones.

- An identifier which is used as both a sanity check and to tell if the region is the global region. When a region is removed, this value is set to a known constant value such that our system can be assured that it is not allocating from a region that has already been removed (this would be an error). We also do not attempt to remove a region that has already been removed. In addition, our system must never try to remove the global region, as it is to last the duration of program execution. Our analysis cannot always tell at compile-time if a region will be the global region or not, so we require this sanity check.

- A pointer to the next free region. This helps our runtime system maintain a list of free regions that can be reused. This value is `NULL` for regions that are in-use.

- A size which reflects how big the region and its array of zone pointers is. This, with the previous bullet, helps our runtime system manage the list of free regions that can be reused when a `CreateRegion()` operation is called.

Our original proof-of-concept discussed in Chapter 3 placed a pointer to the next free byte of a flexipage in the flexipage header. This pointer is redundant and unnecessary. Our system only needs one pointer to the free space per region. Therefore, a modification we have implemented is that this pointer is stored in the region header. This eliminates one pointer from the flexipage header. While this might not be a grand savings of memory, every little bit helps reduce our system’s overall footprint.
5.4.1 Creating a Region

One of the jobs of region transformation is to insert code to create regions just before the points in the program where the region analysis determines that those regions are first needed. The tasks of the code to be inserted are:

- to allocate memory for the header of the new region,
- to initialize all the components of the region header, and
- to return the address of the header.

As shown in Figure 5.4, the size of the region header is a simple function of the number of zones in the region.

In contrast to at least one RBMM system (Mercury), we cannot put the region header at the start of the first flexipage of the region [63]. We cannot do that, because a region with \( n \) zones effectively has \( n \) “first” flexipages. We could pick one, but we would have to treat that one differently from the others (for example, because that flexipage would have room for fewer items than all other flexipages in that zone). We sidestep these problems by allocating the region headers from a memory pool (HeaderPool) that is separate from the pool that supplies the flexipages for zones (PagePool).

When a call to a region creation operation is inserted into the program at compile-time, the number of zones that the region must contain is passed as a static argument. When the region is created at runtime, this value is used to allocate the proper number of zones for the region.

To reduce memory overhead, all zones are created with NULL pointers to their flexipages. When the first request of memory from a zone is made, the zone will request data from our runtime system or from a free-page in our allocator’s free page list (freelist).

The zone must also contain the type information (typeinfo) about the allocations it makes. Such information provides the sizes and offset of structure fields of the data type. Our static analysis generates a typeinfo table that is inserted into the binary and is available at runtime. During zone
initialization at runtime the zone’s typeinfo is set to point to the proper typeinfo in the table.

The following pseudocode outlines how our new CreateRegion() function:

```c
CreateRegion(int n_zones) {
    reg *r = malloc(sizeof(Region) + sizeof(Zone) * n_zones);
    UpdateRegion(r); /* Setup the region’s fields */
    for (int i=0; i<n_zones; i++) {
        /* This will set the typeinfo into the zone */
        r->zones[i] = NewZone(i);
    }
}
```

### 5.4.2 Allocating from a Region

In traditional RBMM systems, each allocation (the equivalent of a call to `malloc`) specifies from what region the new item should be allocated from, by providing a pointer to the header of that region. In our system, the parameter list of the allocation function includes not just a pointer to the region header, but also the zone number that corresponds to the allocated item’s type in that region. From that, the allocation function can look up the zone’s header, and the typeinfo for the type, which gives the size of the item and thus the number of bytes to be allocated. The allocation function gets this number of bytes from the last allocated flexipage of the zone if it has room; if it does not, or if the zone has no allocated flexipages yet, it allocates a new flexipage and adds it to the zone first.

Determining which zones each region must have is an added responsibility of the compile-time region analysis. Note that this must be a global analysis, since different modules may require the inclusion of different types in each region. Further complicating matters, each allocation must specify a single offset in the region structure to find the appropriate zone for that allocation. This offset must be correct for every region that may be used for that allocation, so the offset for each type allocated in a function must
be consistent among all regions that may be used for that allocation in that function. Ensuring this, while minimizing the number of zones in each region, is a complex optimization problem. The implementation discussed in this chapter eases this complexity by requiring every region to contain zones for every type that can belong to a region. This is a gross approximation as it creates many regions with unused zones; however, the approach is easy to implement and allows us to explore the combination of RBMM and GC.

5.4.3 Reclaiming a Region

Reclaiming a region is simple: we release every flexipage in every one of the region’s zones back to \textit{PagePool}, and we release the region header back to \textit{HeaderPool}.

5.4.4 Finding TypeInfos

Since we want to use a type-accurate (non-conservative) collector, we need to be able to find the type of an item from its address. To this end, we maintain a data structure we call the \textit{zone-finder}, which is a variant of the \textit{BIBOP} or \textit{big bag of pages} idea [71]:

- Every item the collector needs to trace on the heap is stored in a flexipage of a zone of a region.

- Both the size and the starting address of every flexipage is an integer multiple of the standard page size. (That is, flexipages are aligned on page boundaries.)

- Conceptually, \textit{PagePool}, the pool from which flexipages are allocated, is a contiguous sequence of pages.

- We pair every page in \textit{PagePool} with a \textit{shadow} word in a new pool, \textit{ZoneFinderPool}, which is the zone-finder.
- If a page in $PagePool$ is not currently in use, then the shadow word corresponding to it will be NULL.

- If a specific page in $PagePool$ is currently in use as the first page of a flexipage in zone $z$ in region $r$, then its shadow word will point to the zone header for $z$. From there, we can reach both the header of region $r$ and the typeinfo describing the type of the items stored in zone $z$ of region $r$.

- If a specific page in $PagePool$ is currently in use as the non-first page of a flexipage in zone $z$ in region $r$, then its shadow word will be a pointer to the shadow word corresponding to the first page of that flexipage, but tagged to indicate that it points to a shadow word rather than a zone header. This occurs because our algorithm for performing address-to-zone mapping treats the high bits of the address as an index into our map. The index calculation works by assuming all pages are of uniform size (currently 4KB). Recall that flexipages must be able to be allocated in sizes larger than 4KB if the program requests a extra-large allocation. Our mapping is aware of this, and will set a bit in the map as mentioned above allowing us to locate the true page start and not potentially the middle of a extra-large flexipage.

Since zone headers and shadow words are both stored at aligned addresses, we use the least significant bit as a tag to distinguish between the last two cases.

Conceptually, $PagePool$ and $ZoneFinderPool$ are arrays with corresponding elements. However, if we want the pools to grow beyond their initially allocated sizes, we must allow them to be stored non-contiguously. For our purposes, pretty much any of the many possible ways of simulating contiguous memory will do. Our implementation represents both $PagePool$ as a sequence of pages, and $ZoneFinderPool$ as a large statically allocated array. The latter is fine for experimental purposes, but is a limitation that
should be lifted to make our system more flexible for real-world use.

5.4.5 Managing Redirections

We garbage collect each zone using a semispace algorithm [27]; that is, we copy every reachable item out of the flexipages currently allocated to the zone (the from-space), into a fresh new set of flexipages (the to-space). When this traversal of reachable items arrives at an item, it needs to know whether that item has been copied to the to-space yet. (Copying a reachable item to the to-space several times would change the aliasing between items, which would be incorrect.) We need one bit per item for this information. These bits are required only during GC, and could thus be kept in temporary data structures, but the management of these data structures would take extra time. To avoid this and to keep the algorithm simple, we reserve space for these bits in each flexipage. The space cost is usually quite small, 1% or less: one bit per item, whose size is virtually always at least 64 bits, and most often 128 bits or more. Therefore the structure of each flexipage is:

- a fixed size flexipage header, which includes the size of the data portion of the page excluding the flexipage header size,
- an array of \( n \) redirection bits, one bit per item,
- any padding required to align the following items, and
- an array of \( n \) items.

The formula for computing \( n \) and the number of padding/alignment bytes before the first item \( bi \) is:

\[
\begin{align*}
  n &= \left\lfloor \frac{(\text{bytes per flexipage} - \text{bytes per header}) \times 8}{1 + (\text{bytes per item} \times 8)} \right\rfloor \\
  bi &= \left\lceil \frac{\text{bytes per header} + \left\lceil \frac{n}{8} \right\rceil}{\text{alignment}} \right\rceil \times \text{alignment}
\end{align*}
\]

where all items begin at an address divisible by \( \text{alignment} \).
Algorithm 1 Preserve data in an item

Require: \textit{base}: The address of the start of an item to preserve
Require: \textit{type}: The type of that item
Require: \textit{fpp}: Points to the flexipage containing that item
Require: \textit{zhp}: Points to header of the zone containing that item

function Preserve(\textit{base, type, fpp, zhp})
    \textit{size} ← \texttt{SizeOf}(\textit{type})
    \textit{newbase} ← \texttt{AllocFrom}(\texttt{ToSpace}(\textit{zhp}), \textit{size})
    \texttt{CopyMemory}(\textit{newbase, base, size})
    \texttt{RedirectBit}(\textit{fpp, base}) ← \texttt{True}
    \texttt{*base ← newbase} \quad \triangleright \text{Set redirect pointer}
return \textit{newbase}

Given the start address of a flexipage, address arithmetic can compute the location of the \texttt{RedirectBit} for an item in that flexipage, and vice versa.

Between two GC cycles, each \texttt{RedirectBit} in each flexipage contains a value of zero. When the traversal encounters a reachable item whose \texttt{RedirectBit} is zero, it copies the item to the to-space, and sets its \texttt{RedirectBit} to a value of one. To let later parts of the traversal know not just that the item \textit{has} been copied but also \textit{where} it has been copied to, the traversal also records the address of the item in to-space in the first word of the item. (It is okay to overwrite any part of the user data stored in the old copy of the item, since it will not be referred to anymore.) All this is shown in Algorithm 1.

Of course, this assumes that all items are big enough to hold a pointer. This is why our system allocates a word (the size of a pointer) even for requests that ask for less memory than that. It is not alone in this; virtually all other memory management systems do the same, including the usual implementations of \texttt{malloc}.

When a GC cycle is complete, all the pages of all the flexipages of the collected regions are returned to the freelist of \texttt{PagePool}. Before any flexipage is reused, all its bits will be set to zero, including its redirection bits.
5.4.6 Collecting Garbage

Our region-aware garbage collector is setup to perform a collection once a specific number of bytes has been allocated from the system. Since our RBMM system manages memory, we know when a program has allocated this much memory. This memory limit is an increasing value and is initially set to twice the default page size. We define our default page size as being 4096 bytes, which is a common page size for a 64-bit Linux kernel. Once a GC completes, the upper-bound is doubled. For example, the first GC will commence once 8192 total bytes have been requested. The next GC will trigger when double that value is reached, in this case 16384 bytes, and so on. This, perhaps conservative, strategy allows us to better study the impact of our region-aware garbage collector. Studying alternative collection trigger strategies could prove useful; however, that is reserved for future implementations to study.

The top level of our GC algorithm is shown in Algorithm 2. This algorithm is the main driver of our GC and is responsible for determining if a particular region can be collected from or not. This routine is also responsible for swapping the to and from space pointers for the zones that are collected from.

The first parameter to the GC algorithm is the root set, i.e. the set of all the registers, stack slots and global variables that may contain pointers to items in regions. (We start by making copies of the original register values in memory, and copy the possibly-directed values back to the registers when we are done.) The second parameter specifies the set of regions from which this invocation of the collector should recover memory. Currently we collect from all regions; however we explain a more advanced algorithm below which permits a subset of all regions to be collected from. This set need not be the set of all regions. If the runtime system responsible for controlling the collection process expects that some regions have very little garbage, it can omit them from GC_regions. A region left out of GC_regions will still be traversed (traced) by our algorithm if such traversal may lead to reachable
items in regions which are in $GC_{\, regions}$, but

- the collector will not need space to store copies of all the reachable items in those regions, reducing memory requirements when those requirements are otherwise at their peak, and

- the collector will not need to spend any time copying all the reachable items in the regions to the to-space, and updating all the pointers to the moved items.

The algorithm starts by recording, in each region header, whether the region is being collected in this collection, and whether it needs to be traced.

After that, Algorithm 2 finds all items in the collected regions that are reachable from the roots, using Algorithms 3 and 4, which we discuss below. Together these algorithms preserve each reachable item in a collected region by copying it from its original location in a from-space flexipage of one of the region’s zones to the to-space of that zone, which consists of its own list of flexipages.

Once all reachable items have been copied, and the pointers to them updated to point to the new copies, the algorithm releases the memory occupied by the zones’ original set of flexipages. In other words: when collection is complete, the from-space pages are reclaimed/freed by returning them to the free-page list so that they can be reused for other zones as needed.

Algorithm 3 locates reachable items in the regions being collected and copies them to the to-space of their zone. This algorithm maintains the invariant that any traced item that has its redirect bit set does not need to be traced again. This prevents tracing a cyclic item forever. Another invariant is that all items that are collected will have their redirect bit set. This ensures that all pointers to collected items will always refer to the copied version in the to-space after GC completes.

The $\text{PreserveAndTrace}$ function is invoked not with the address of the item it is to preserve and trace, but with a pointer to that address, so
that if and when it needs to move the item, it can update the address that pointed to it. When it is invoked, *addrptr* will point either to a root (such as a global variable or a stack slot containing a pointer), or to a part of the heap that itself contains a pointer. The pointers to roots supplied by Algorithm 2 always point to the start of a root item, as promised by the *toplevel* = *True*; pointers supplied by tracing may point inside (i.e. not at the start of) items, as allowed by *toplevel* = *False*. For example, a pointer field within a structure might not be the first item (start) of the structure, but could be in the middle of it somewhere. Since we do not have per-field redirect bits, our system can only detect if the whole object has been redirected or not.

The value of *addr* may or may not point into the heap, which in our case means “into one of the regions.” If it does, then we can use the data structures described in Section 5.4.4, represented here by the function `LookupHeap`, to find out the address of the flexipage containing the item at *addr*. From that, the function can use address arithmetic to compute *base*, the address of the start of the item (*addr* may point into the middle

---

**Algorithm 2** Garbage collect from regions

**Require:** *Roots*: The set of root variables

**Require:** *GC_regions*: The set of regions to collect

**function** `GC(Roots, GC_regions)`

```plaintext
  for all `rhp ∈ all_regions` do
    RegionBeingCollected(`rhp`) ← `rhp ∈ GC_regions`
    RegionNeedsTracing(`rhp`) ← some region in `GC_regions` is reachable from `rhp`

  for all `root ∈ Roots` do
    PreserveAndTrace(`root`, True)
  end

  for all `rhp ∈ GC_regions` do
    for all `zhp ∈ ZonesOf(rhp)` do
      Free(FromSpace(`zhp`))
      FromSpace(`zhp`) ← ToSpace(`zhp`)
      ToSpace(`zhp`) ← nil
    end
  end
```

---
Algorithm 3 Preserve an item and everything it can reach

Require:  \textit{addrptr}: Pointer to the address of an item

Require:  \textit{toplevel}: Is the call coming from Algorithm 2?

function \textsc{PreserveAndTrace}(\textit{addrptr, toplevel})

\begin{verbatim}
addr ← \ast addrptr
if addr is in the heap then
    \langle fpp, base, zhp \rangle ← \textsc{LookupHeap}(addr)
    type ← \textsc{TypeIn}(zhp)
    offset ← addr − base
    rhp ← \textsc{ContainingRegion}(zhp)
    if \neg \textsc{RegionBeingCollected}(rhp) then
        newbase ← base  \triangleright Item is not moved
        needstrace ← \textsc{RegionNeedsTracing}(rhp)
    else
        if \neg \textsc{RedirectBit}(fpp, base) then
            newbase ← \textsc{Preserve}(base, type, fpp, zhp)
            \*addrptr ← newbase + offset
            needstrace ← \textsc{RegionNeedsTracing}(rhp)
        else
            newbase ← *base  \triangleright Get redirect pointer
            \*addrptr ← newbase + offset
            needstrace ← False  \triangleright Has been traced already
    else if addr is not null then
        \triangleright if addr is not in the heap, it must refer to a root
        if toplevel then
            newbase ← addr  \triangleright Top level refs point to the start
            type ← \textsc{.TypeOf}(addr)
            needstrace ← True
        else
            needstrace ← False  \triangleright A top level call will trace it
    if needstrace then
        \textsc{Trace}(newbase, type)
\end{verbatim}

\end{verbatim}

of the item). If \(s\) is the size of the items in the flexipage, then

\[
base = fpp + bi + s \times \left\lfloor \frac{addr - (fpp + bi)}{s} \right\rfloor
\]
We need to know *base* because if we copy the item, we must copy *all* of it. If the item ends up moved, the updated pointer must point to the same offset within the item as it did before.

Given the flexipage pointer, **LOOKUPHEAP** can also use the zone-finder to find the identity of the zone containing the flexipage. We can then follow the pointer in the zone header to the header of the region containing it. If this region is not being collected, then the item will survive the collection, at its current address, without us doing anything (though we may still need to trace any pointers inside the item). If this region is being collected, then Algorithm 2 will free all the flexipages of all the zones of the region, and we must copy the item to the corresponding to-space, unless this has already been done. If the redirect bit says that it has not yet been done, then we call **Preserve**, the function in Algorithm 1, to copy it to the zone’s to-space. **Preserve** returns the new address of the item, and we set the original pointer to the item to refer to the original offset from this new address. **Preserve** also records both the fact that the copying has been done (by setting the redirect bit corresponding to this item in its flexipage) and the address of the new home of the item (in the first word of the item). So the next time the traversal reaches this item, the redirect bit will tell us that we do not need to copy the item again, and that we can instead pick up the new address of the item from the first word in its old copy. In this case, the traversal will also have traced all the pointers inside the item, so we need not process them again. We can similarly skip the processing of the pointers inside the item if the item is in a region from which no region being collected can be reached either directly or indirectly.

Since we do not garbage collect the places that may contain roots, *i.e.* the stack, the global variables, and the registers, we need not concern ourselves with protecting any item that is not in the heap against being moved. Since Algorithm 2 will eventually invoke Algorithm 3 on every root, we need not trace roots when we reach them by following pointers in items. This is just as well, since those pointers may point inside roots that are structures, and
Algorithm 4 Trace an item and preserve all items it can reach

Require: base: Pointer to the start of the item
Require: type: The type of the item located at base

function Trace(base, type)
    for all aioff ∈ AddrsInside(type) do
        aiaddr ← base + aioff
        PreserveAndTrace(aiaddr, False)
    end for
end function

finding the starts of those structures would be far from trivial.

The last step of Algorithm 3 is to invoke Algorithm 4 on roots and items on the heap that (a) may contain pointers that lead, directly or indirectly, to reachable items in the regions being collected, and (b) are not known to have been traced before. The traced items may be pointers, for which the AddrsInside function should return the offset 0. Or they may be structures containing pointers, for which it should return the offset of all the pointer-valued fields inside the structure, whether they are fields of the structure itself or of its parts. We then traverse the items all these pointers point to.

Note that we trace the pointers in the version of the item in the to-space, not the from-space. That is because in the from-space, the first word of the item will have been overwritten by the redirect pointer (also called the forwarding pointer).

Figure 5.5 shows an example illustrating these algorithms. Figure 5.5(a) shows part of the memory as GC begins: the stack contains two pointers, xptr and yptr, that point to two structures in the same zone, which has one flexipage. These structures each contain one non-pointer, whose contents are irrelevant here, and one pointer, which in this case point to each other, so this is a cyclic data structure. Note the flexipage contains two garbage structs, and the redirection bits are all 0.

After Algorithm 2 determines which regions to collect, we process the root set. We start by calling PreserveAndTrace with xptr. The struct xptr points to ("item 1") is on the heap, in a region to be collected, and the redirect bit for it is clear, so we Preserve it: we copy it to to-space, set
Figure 5.5: Example: copying two items to to-space
its redirected bit in from-space, and overwrite the first word of the struct in from-space with a pointer to the new copy in to-space. We then update xptr to point to the new copy as well. This is shown in Figure 5.5(b).

Next we call PreserveAndTrace on the sole pointer in the freshly copied struct. Again this points to a struct (“item 2”) on the heap, in a region to be collected, whose redirect bit is clear, so we Preserve it, and update the pointer we followed (in item 1) to point to the new copy. Next we Trace the newly preserved struct, but this time the sole pointer points to a struct (item 1 again) whose redirected bit is now set. In this case we do not preserve or trace the struct, we just look up its new location in to-space, so we can use it to overwrite the pointer in item 2 (which used to point to item 1 in from-space).

When Algorithm 2 calls PreserveAndTrace on the second root, yptr, it finds that the struct it points to, item 2, already has its redirect bit set. It will therefore update yptr to point to the new location of item 2 in to-space, but will not trace item 2 again. This will leave the state shown in Figure 5.5(c). As you can see, copy collection can eliminate internal zone fragmentation. The garbage that existed in the from-space flexipage has been eliminated and only reachable data exist contiguously in the to-space.

5.5 Managing Arrays and Slices

The Go language provides arrays, pointers to arrays, and slices. Arrays are fixed-size contiguous collections, and array pointers refer to fixed-sized collections as well, since the type of array pointers includes the number of elements in the array as well as the type of the elements. On the other hand, while the compiler knows the type of the elements of a slice, it does not know their number; the size of a slice is dynamic. The Go implementation represents each slice as a structure holding a reference to some element of an array, as well as a capacity (the number of elements in the slice, starting at the pointed-to element), and a count (the number of initial elements in the
slice that are meaningful). Thus, slices are simply Go structures comprising three members, and, except in the optimization we describe in Section 5.5.2, we treat them as such.

### 5.5.1 Finding the Start of an Array

Some slices will point to elements in the middle of the target array. The `PreserveAndTrace` function needs to know the start address of the item to be preserved. With scalar items, once we know the start address of a flexipage, we can use address arithmetic to convert the address of any part of an item into the address of the start of the item.

We want to prevent duplicating of data when collecting from arrays. If the data has already been collected (copied), there is a chance that a pointer into the middle of the array exists and that copying this middle element of the array would result in duplicated data. This occurs because the array was copied as a single contiguous element and not as a collection of individual elements. Therefore, no redirected bits for the elements exist, and duplicate data could result. As we mentioned earlier, duplicates are dangerous, since pointers to any of the duplicates would not be guaranteed to have an accurate picture of the state of the system. For instance, if one of the duplicate data pieces were to be modified, potentially a subset of the pointers would see this change, but not all of pointers.

Consider the array and pointer into it depicted in Figure 5.6. Since our collector does not specify the order from which root variables are to be traced, it is perfectly fine for our collector to trace `p` first and then `a`. From
the latter figure, our collector would first trace a and then p. Since a points
to the head of the array, it can access all elements, therefore the collector
must preserve the entire array. The collector will eventually scan p. Since p
points at an element located within the array, our collector should not copy
the element, as it was previously copied due to a being collected. If the data
at p was copied, it would result in a duplicate as illustrated in Figure 5.7.
In the this figure, if p is modified, it would not be seen by any variables
accessing element 50 from a.

We handle arrays specially, by placing all arrays of a given type (regardless
of length) into a single zone dedicated to arrays of that type. However,
this also means that we cannot find the start of an array by address calcu-
lation. Our solution is to prefix each array with a small header containing
just its size. (Most memory management systems do this for every item;
we do it only for arrays.) When tracing a pointer to or into an array, we
can look up its address in the zone finder, which will give us a pointer to
the start of the flexipage containing the array item. The first item in the
flexipage starts just after the flexipage header. Given the start address of
an item, i.e. the address of its array header, the size allows us to calculate
the address of the start of the next item in the flexipage, if there is one.
So we can find the start address of the array that a pointer points into by
traversing through the array items on the flexipage. The address we want
is the last item start address we encounter in this traversal that is smaller
than the pointer’s value.
5.5.2 Preserving Only the Used Parts of Arrays

Our proof-of-concept evaluated later only preserves entire slices and arrays, and does not consider the individual elements. However, we now present a design for garbage collecting a subset of an array, which might prove helpful to future implementers.

Our PreserveAndTrace algorithm treats any reference to any part of an item as a reference to the entire item. A reachable variable whose type is an array or array pointer keeps all the elements alive. For slices, however, we can do better, provided live slices refer only to a part of the array that the slices were derived from, and there are no reachable references to the whole array. In such situations, we can reclaim the unneeded elements in such arrays, if we can modify the algorithm we use to trace slice headers. First we present how we handle slices; later we will return to discuss how array and array-pointer-valued variables fit into our scheme.

The Go language semantics requires that if an array and slice, or two slices, shared the memory of some elements before a collection, they must also share those same elements after the collection. We therefore cannot copy reachable array elements individually; we must ensure that contiguous sequences of reachable array elements are copied to a new (possibly smaller) array in which they are still contiguous.

When tracing arrives at a slice header, we know that the array elements referred to by the slice are reachable. Unfortunately, we cannot know at that time whether the elements before and after these in the array are reachable or dead: it is possible that they have not yet been visited by the collector, but will be visited later. In general, we can know which elements of an array are reachable and which are dead only once a GC has finished tracing all reachable data.

We could add an extra pass to the end of every collection, and defer the copying of array slices until this pass. However, this extra pass would add significant overhead, because not preserving an array when tracing it would reduce locality, and because we would need extra data structures to...
keep track of the deferred work. These data structures would occupy space during each collection, *exactly when free space is scarcest*. We therefore choose to use a conservative approximation: when we get to an array, we copy to to-space the set of array elements that were reachable at the end of the last collection. This means that an array element that becomes dead will survive one collection, but not two.

This optimization needs more information attached to each array item than what we would need in its absence, which we just described in Section 5.5.1. This information consists of:

- **ARRAYNumBytes**, the number of bytes occupied by the item (as before).
- **ARRAYNumElts**, the number of elements in the array; redundant, as it could be computed from **ARRAYNumBytes**, but storing it avoids unnecessary recomputations.
- **SEENPrev**, an array of **ARRAYNumElts** bits. The bit is 1 iff the corresponding element was reachable at the end of the last collection. Initialized to 1 when the array is first created.
- **SEENCurr**, an array of **ARRAYNumElts** bits. The bit is 1 iff the corresponding element is reachable during the current collection. Always initialized to 0; meaningful only during a collection.
- **Elements**, the elements of the array themselves.

Our optimization modifies Algorithm 3 so that when the collector traces a slice, it will invoke Algorithm 5 instead of Algorithm 4. This algorithm has a chance to avoid preserving unneeded elements of the array holding the slice’s elements, but only if the array is stored on the heap. If it is stored in the executable’s read-only `.rodata`, `.data` or `.bss` sections, then the array must be a global variable. This means that we need not take action to preserve its storage, and since all global variables are roots, that root either has already
been or will be traced later, as an array (not as a slice). The array cannot
be on the stack, because the Go compiler performs escape analysis, and this
changes the storage class of any function-local array that a slice may ever
refer to, converting it from stack allocated to heap allocated.

If the array holding the slice’s data is on the heap, we use the algorithms
of Section 5.5.1 (represented by function \texttt{LOOKUPHEAPARRAY}) to find the
address of the flexipage storing the item, and from that, the start of the
array item, and the zone containing that flexipage. From the addresses of
the first elements of the slice and of the array, we can calculate \textit{fse}, the
index of the first slice element in the array. From that and the capacity of
the slice, we can calculate \textit{lse}, the index of the last slice element in the array.

Consider the situation when Algorithm 5 is invoked on slice header S1 in
the example in Figure 5.8. This slice has a capacity and count of 2, and its
data pointer points to the element at index 3 in the array. We will thus set
\textit{fse} to 3 and \textit{lse} to 4. However, we cannot copy to to-space just the subarray
containing only elements 3 and 4. For example, if we later see a reference
to slice S2, which also has a capacity and count of 2, but its data pointer

![Figure 5.8: Copying only the previously reachable parts of arrays](image-url)
points to the element at index 2 in the array, the copies of the two slices must share the element corresponding to the index 3 in the original array.

The sequence of elements we may need to have contiguous in the copy is restricted to the neighboring elements that were reachable during the last collection. In our example, the SEENPREV bit vector for the array has a 1 in every position except the ones at indexes 0 and 6, so the first bit in the contiguous sequence of 1 bits that includes the 1 bits at positions 3 and 4 is at index $fce = 1$, and the last bit in that contiguous sequence of 1 bits is at index $lce = 5$. This is why we want to make sure that there is a copy in to-space of the subarray consisting of elements 1 to 5.

If all of the bits in SEENCURR starting from $fce$ to $lce$ are 0s, then the subarray has not yet been copied to to-space, so we do the copying then and there. After figuring out the amount of memory needed for the new array item, we reserve memory for it in to-space. We then fill in the array item’s header, its SEENPREV and SEENCURR arrays, and finally the elements, which we copy from the original array. The SEENCURR array has all its bits set to 0s: those bits will be meaningful in the next collection, not this one. We set the SEENPREV bits in the to-space copy only for the array elements that this slice refers to, since so far these are the only elements that we know are reachable.

After we copy all the elements of the subarray from from-space to to-space, we overwrite the first word in the first element copied in from-space with the address of the copy in to-space. This ensures that later calls to TRACESLICE arriving at this subarray (e.g., when TRACESLICE is invoked with S2) will know where the copy is.

Once we have preserved the data in the contiguous elements, we need to trace any pointers in the meaningful part of the slice. So we iterate over all those elements, tracing pointers in elements we have not traced before. Note that we set the SEENPREV bit corresponding to such items even if this call to TRACESLICE did not copy the subarray. In our example, this will happen when tracing the copy of element 2 during the invocation of
TraceSlice for slice S2. This will tell the next invocation of the collector that the elements at indexes 1, 2 and 3 are reachable in the copied subarray; these correspond to indexes 2, 3 and 4 in the original array.

The elements in slices that correspond to the difference between the Count and the Capacity (if there is one) do not contain data that the program may use, but they must be there, contiguous with the earlier elements, in case the program expands the slice. If there is a slice S3 that has a count of 2 but a capacity of 4, and its data pointer points to the element at index 2 in the array, then we must mark index 4 in the copy (index 5 in the original) as seen, because if we did not, then the collection after the next one would recover its memory, which would prevent the correct operation of any expansion operations on S3.

Since the body of the function after the initial test can rely on the capacity of the slice being at least one, one of the loops that together iterate i from 0 to cap will set the SeenCurr bit for fse. Since fse is guaranteed to be in the range fce..lce, all later invocations of TraceSlice on a slice that fits in that range will know that the subarray in that range has already been copied to to-space.

Just as our optimization must ensure that we call TraceSlice instead of Trace when tracing a slice header, we must handle two other cases specially as well. The first is arrays on the heap, or pointers to them. For these, we need to invoke a version of TraceSlice that acts as if it was tracing a slice whose count and capacity are both the array size (in Go, this is available as part of the type of both arrays and pointers to arrays), the only differences being that (1) the capacity and count come from somewhere else, and (2) relocation must be reflected by an assignment to something other than Data(shp). The second case is pointers to values that happen to point to or inside an array element. We can handle these as if we were looking at an one-element slice, though recording the relocation must be done differently yet again.

Since array elements can be of any type, the criterion that tells Algo-
rithm 3 that it should call not TRACe but TRACeSLICE (or its equivalents for arrays and array pointers) should not be the type of the item being traced, but a property of the zone that contains it. The obvious property to test is “does this zone contain array data.” However, the approach we described in this section adds both space and time overheads. If arrays in a zone typically die all at once, then we would not want to incur these overheads, because they would not pay for themselves through the earlier recovery of the memories of array elements. If either the programmer or a profiling system can predict which zones fall into which category, they can control whether the algorithms of this section are applied to each zone by including a bit in the headers of zones containing arrays that tells the algorithms operating on the zone’s flexipages, including Algorithm 3, which item representation the zone uses, and therefore whether they should call TRACeSLICE, or just a version of TRACe adapted to the simpler data structures described in Section 5.5.1.

5.6 Handling Other Go Constructs

Go provides several features we have not yet discussed. We defer the discussion of go-routines until Chapter 6. While we have not modified our collector to handle the constructs in this section, we do provide discussion as to how they can be dealt with in future implementations.

Interface types in Go might be expected to present something of a problem, since values declared in a function as having an interface type actually have some other type which is not known when the function is compiled. However, our scheme handles interface types without adaptation. User code that deals with the item without knowing its actual type, knowing only what interface it implements, never needs to know what zone the item is stored in. However, when an instance of an interface type is created, its actual type must be known, so it will naturally be placed in the correct zone. When tracing an interface type item, our functions will find the flexipage and the zone it occurs in, and will determine its actual type from that before preserving
Algorithm 5 Trace a slice header, and preserve and trace its slice

Require: \texttt{shp}: Address of the slice header

\begin{algorithm}
\caption{TraceSlice(\texttt{shp})}
\begin{algorithmic}
\Function{TraceSlice}{\texttt{shp}}
\If{\texttt{Capacity(\texttt{shp}) = 0}}
\Return
\EndIf
\State \texttt{slicestart} \leftarrow \texttt{Data(\texttt{shp})}
\If{\texttt{slicestart} is in the heap}
\State \langle \texttt{base, zhp} \rangle \leftarrow \texttt{LookupHeapArray(slicestart)}
\State \texttt{type} \leftarrow \texttt{ElementType(TypeIn(zhp))}
\State \texttt{es} \leftarrow \texttt{SizeOf(type)} \quad \triangleright \text{Element size}
\State \texttt{cap} \leftarrow \texttt{Capacity(\texttt{shp})}
\State \texttt{fse} \leftarrow (\texttt{slicestart} \& \texttt{Elements(base, 0)})/\texttt{es}
\State \texttt{lse} \leftarrow \texttt{fse} + \texttt{cap} - 1
\State \texttt{fce} \leftarrow \texttt{fse}
\While{$0 \leq \texttt{fce} - 1 \land \texttt{SeenPrev(base, fce - 1)}$}
\State \texttt{fce} \leftarrow \texttt{fce} - 1
\EndWhile
\State \texttt{lce} \leftarrow \texttt{lse}
\While{$\texttt{lce} + 1 < \texttt{cap} \land \texttt{SeenPrev(base, lce + 1)}$}
\State \texttt{lce} \leftarrow \texttt{lce} + 1
\EndWhile
\State \texttt{copybase} \leftarrow \texttt{PreserveElements(slicestart, fce, lce, fse, lse, es)}
\State \texttt{Data(\texttt{shp})} \leftarrow \&\texttt{Elements(copybase, fse - fce)}
\State \texttt{count} \leftarrow \texttt{Count(\texttt{shp})}
\For{$i \leftarrow 0 \to \texttt{count} - 1$}
\If{\texttt{SeenCurr(base, fse + i) = 0}}
\State \texttt{SeenCurr(base, fse + i)} \leftarrow 1
\State \texttt{SeenPrev(copybase, fse - fce + i)} \leftarrow 1
\State \texttt{eltbase} \leftarrow \&\texttt{Elements(copybase, fse - fce + i)}
\For{all \texttt{aioff} \in \texttt{AddrsInside(type)}}
\State \texttt{aiaddr} \leftarrow \texttt{eltbase} + \texttt{aioff}
\State \texttt{PreserveAndTrace(aiaddr, False)}
\EndFor
\EndIf
\EndFor
\For{$i \leftarrow \texttt{count} \to \texttt{cap} - 1$}
\State \texttt{SeenCurr(base, fse + i)} \leftarrow 1
\State \texttt{SeenPrev(copybase, fse - fce + i)} \leftarrow 1
\EndFor
\EndFunction
\end{algorithmic}
\end{algorithm}

147
Algorithm 6 Preserve slice elements

Require: \textit{slicestart}: Starting address of slice data
Require: \textit{fce}: Index of the first contiguous element
Require: \textit{lce}: Index of the last contiguous element
Require: \textit{fse}: Index of the first slice element
Require: \textit{lce}: Index of the last slice element
Require: \textit{es}: The size of each element

\noindent \textbf{function} \textsc{PreserveElements}(\textit{slicestart, fce, lce, fse, lse, es})

\noindent \langle \textit{base, zhp} \rangle \leftarrow \textsc{LookupHeapArray}(\textit{slicestart})

\noindent \textbf{if} $\forall i \in \textit{fce}..\textit{lce} . \neg\textit{SeenCurr(\textit{base}, i)}$ \textbf{then}

\noindent \textit{numelts} $\leftarrow \textit{lce} - \textit{fce} + 1$

\noindent \textit{copybytes} $\leftarrow \left\lceil \frac{\textit{headerbytes} + \left\lceil \frac{(2 \times \textit{numelts})}{8} \right\rceil}{\textit{alignment}} \right\rceil \times \textit{alignment} + \textit{numelts} \times \textit{es}$

\noindent \textit{copybase} $\leftarrow \textsc{AllocFrom(ToSpace(\textit{zhp}), \textit{copybytes})}$

\noindent \textit{ARRAYNumBytes(\textit{copybase})} $\leftarrow \textit{copybytes}$

\noindent \textit{ARRAYNumElts(\textit{copybase})} $\leftarrow \textit{numelts}$

\noindent \textbf{for} $i \in 0..\textit{numelts} - 1$ \textbf{do}

\noindent \textit{SeenPrev(\textit{copybase}, i)} $\leftarrow \textit{fse} \leq \textit{fce} + i < \textit{lse}$

\noindent \textit{SeenCurr(\textit{copybase}, i)} $\leftarrow 0$

\noindent \textbf{COPYMemory(\&\textsc{Elements}(\textit{copybase}, 0),

\noindent \textit{slicestart, \textit{numelts} \times \textit{es}})

\noindent \textit{*(\&\textsc{Elements}(\textit{base}, 0) + \textit{fce} \times \textit{sz})} $\leftarrow \textit{copybase}$

\noindent \textbf{else}

\noindent \textit{copybase} $\leftarrow \textit{*(\&\textsc{Elements}(\textit{base}, 0) + \textit{fse} \times \textit{sz})}$

\noindent \textbf{return} \textit{copybase}

and tracing it.

An interface type in effect stands for all the types that implement all the methods of that interface. In some cases, this may lead to regions with many zones, one for each actual type that is passed to a function or a set of functions expecting an interface type, with many of these zones containing very few or no items at all. In such cases, much space will be wasted on flexipages with few inhabitants. An alternative approach would have the region inference algorithm put items that are used as values of interface types into a special zone in each relevant region, a zone in which each item
contains a tag. This would trade slower GC and higher per-item memory overhead for a lower per-actual-type overhead. Determining which of these two approaches is better, and under what circumstances, is a matter for future work.

There are aspects of the Go implementation, such as strings and maps, that use specialized data representations. The algorithms that we have presented in this chapter need minor adaptations to handle these representations.

5.7 Evaluation

In order for us to measure how well our RBMM system performs with our region-aware garbage collector, we executed a series of tests to measure time and memory usage. Most of the benchmarks are derivatives of those mentioned in Chapter 3 with some parameters adjusted in order to increase runtime. Due to a buggy implementation, we were only able to run a subset of the tests from the Debian Language Shootout benchmark suite provided in the GCC test suite for Go. Further limiting our test set is the fact that our newer modification does not support multiple module Go programs. Therefore, only single module tests have been considered. It just happens to be that all of the Shootout tests are single module, but also very small in terms of lines of code. Once again, we chose programs that did not require the use of Go routines. With the exception of matmul,\(^1\) we only show results for benchmarks which we can verify output from.

The additional benchmark programs we added were also from the Debian Language Shootout and range from approximately 58 to 85 lines of code. These programs are fannkuch (integer manipulation program), mandelbrot (fractal generator), and pidigits (pi digit calculator).

Benchmarking tests were conducted on the same machine as our original

\(^1\)This benchmark provides interesting information even though it failed verification when using our region-aware garbage collector.
series of evaluations (see Chapter 3). However, given the vast amount of
time between the latter chapter and the current one, some software updates
have been implemented on the test machine. For instance, the OS running
the tests has now been upgraded to Ubuntu 13.04. Similarly, the GCC
used to compile both our modifications (our plug-in and runtime system)
and the tests themselves is version 4.7.2 (which supports the Go version 1.0
specification). The GCC Go libraries used during runtime were provided by
the same GCC 4.7.2 and are from Go version 1.01. Some of our modified
runtime system is based on code from GCC’s libgo version 4.7.1 and 4.7.0.

5.7.1 Methodology

The results reported here involved running each test program with and with-
out output. We verified that our modifications do not alter the output from
that of which the unmodified program will produce. The performance results
for each test were gathered by executing each benchmark with its output
disabled. This allows us to measure execution time more precisely without
the additional OS overhead of printing values to the display.

Both the high water mark (HWM), which represents the maximum resi-
dent set size of the process, and the elapsed time, are reported as an average
over ten runs for each benchmark.

The *Inuse Pages* and *Free Pages* columns represent the number of flex-
ipages our system has allocated and whether they are currently being used
to produce allocations from, or if they exist on a free list and are available
when a new page is needed. All dynamic memory provided to the program
and for our runtime system are reflected in these page metrics.

5.7.2 Results

Table 5.1 contains our benchmarking results. Of interest are both the time
and space usage of the benchmarks. Unmodified tests are labeled as *(plain)*.
At the time of experimenting, Go’s existing collector was a mark-sweep
<table>
<thead>
<tr>
<th>Test</th>
<th>HWM (MB)</th>
<th>Elapsed (seconds)</th>
<th>Inuse Pages</th>
<th>Free Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>N</td>
<td>KB</td>
</tr>
<tr>
<td>binary-tree (plain)</td>
<td>87.55</td>
<td>11.41</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>binary-tree (rbmm)</td>
<td>63.71</td>
<td>2.87</td>
<td>9,364</td>
<td>37,456.0</td>
</tr>
<tr>
<td>binary-tree (rbmm+gc)</td>
<td>513.29</td>
<td>4.19</td>
<td>9,420</td>
<td>37,680.0</td>
</tr>
<tr>
<td>fannkuch (plain)</td>
<td>8.37</td>
<td>7.76</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>fannkuch (rbmm)</td>
<td>8.39</td>
<td>7.89</td>
<td>4</td>
<td>16.0</td>
</tr>
<tr>
<td>fannkuch (rbmm+gc)</td>
<td>8.43</td>
<td>7.59</td>
<td>6</td>
<td>24.0</td>
</tr>
<tr>
<td>mandelbrot (plain)</td>
<td>8.43</td>
<td>4.66</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>mandelbrot (rbmm)</td>
<td>8.45</td>
<td>4.72</td>
<td>0</td>
<td>0.0</td>
</tr>
<tr>
<td>mandelbrot (rbmm+gc)</td>
<td>8.48</td>
<td>4.66</td>
<td>0</td>
<td>0.0</td>
</tr>
<tr>
<td>matmul (plain)</td>
<td>84.92</td>
<td>14.44</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>matmul (rbmm)</td>
<td>78.72</td>
<td>11.99</td>
<td>6,005</td>
<td>72,100.0</td>
</tr>
<tr>
<td>matmul (rbmm+gc)</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>meteor-contest (plain)</td>
<td>8.60</td>
<td>0.18</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>meteor-contest (rbmm)</td>
<td>8.67</td>
<td>0.12</td>
<td>3</td>
<td>12.0</td>
</tr>
<tr>
<td>meteor-contest (rbmm+gc)</td>
<td>12.52</td>
<td>0.12</td>
<td>5</td>
<td>20.0</td>
</tr>
<tr>
<td>pidigits (plain)</td>
<td>9.56</td>
<td>44.74</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>pidigits (rbmm)</td>
<td>9.56</td>
<td>44.67</td>
<td>0</td>
<td>0.0</td>
</tr>
<tr>
<td>pidigits (rbmm+gc)</td>
<td>9.57</td>
<td>44.68</td>
<td>0</td>
<td>0.0</td>
</tr>
</tbody>
</table>

Table 5.1: Performance results

collector. Tests using RBMM with Go’s garbage collector are labeled as *(rbmm)*, and tests utilizing RBMM and our region-aware garbage collector are labeled as *(rbmm+gc)*.

We have not optimized our garbage collector. Much time was spent debugging and correcting issues such that we could have this small benchmark set run. Building such a system is not trivial; however, we do feel that we have enough in place to perform measurements which can be helpful for future researchers.

An interesting measure to consider when reading these results is the additional size required to support the garbage collector. Such “bookkeeping” data includes the *typeinfo*, global variable, stack, and register information tables that our collector uses to scan allocated items and to traverse the call stack. Both of our RBMM systems create regions dynamically at runtime.
Table 5.2: Binary sizes of the benchmarks

<table>
<thead>
<tr>
<th>Test</th>
<th>Executable Size (KB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>binary-tree (plain)</td>
<td>66</td>
</tr>
<tr>
<td>binary-tree (rbmm)</td>
<td>147</td>
</tr>
<tr>
<td>binary-tree (rbmm+gc)</td>
<td>211</td>
</tr>
<tr>
<td>fannkuch (plain)</td>
<td>65</td>
</tr>
<tr>
<td>fannkuch (rbmm)</td>
<td>145</td>
</tr>
<tr>
<td>fannkuch (rbmm+gc)</td>
<td>204</td>
</tr>
<tr>
<td>mandelbrot (plain)</td>
<td>65</td>
</tr>
<tr>
<td>mandelbrot (rbmm)</td>
<td>143</td>
</tr>
<tr>
<td>mandelbrot (rbmm+gc)</td>
<td>208</td>
</tr>
<tr>
<td>matmul (plain)</td>
<td>66</td>
</tr>
<tr>
<td>matmul (rbmm)</td>
<td>146</td>
</tr>
<tr>
<td>matmul (rbmm+gc)</td>
<td>215</td>
</tr>
<tr>
<td>meteor-contest (plain)</td>
<td>78</td>
</tr>
<tr>
<td>meteor-contest (rbmm)</td>
<td>157</td>
</tr>
<tr>
<td>meteor-contest (rbmm+gc)</td>
<td>247</td>
</tr>
<tr>
<td>pidigits (plain)</td>
<td>4,499</td>
</tr>
<tr>
<td>pidigits (rbmm)</td>
<td>4,462</td>
</tr>
<tr>
<td>pidigits (rbmm+gc)</td>
<td>4,458</td>
</tr>
</tbody>
</table>

Table 5.3: Number of garbage collections

For the rbmm+gc tests, our system allocates additional memory upon each region creation to contain its zone header information. Recall that our zones are only useful for GC, therefore in the rbmm tests, each region contains just a single type agnostic zone.

Table 5.2 lists the binary sizes for all three versions of each program.
described above without the statistics gathering code compiled in. These numbers reflect the additional memory a test requires when the test is initially loaded into main memory. This size will impact the minimum bound on the HWM values depicted in the former table. Of course, not all of a binary is loaded into memory. Metadata stored in the file will be used by the OS to properly load the executable into memory. RBMM alone contributes an average of 79.50 KB to the binary’s size. Adding in our region-aware garbage collector contributes, on average, an additional 68.33 KB over that of just RBMM alone. Note that these values for rbmm+gc also reflect additional debugging information for aiding GC debugging. While these data are not used during runtime, they were not removed due to complications.

Our system never returns memory to the OS. Instead, the system recycles memory for later allocations. When looking at the HWM values, the In Use Pages and Free Pages metrics should also be taken into consideration. These values show the data that only our runtime systems (rbmm and rbmm+gc) know of. The runtime memory footprint results, as seen in the HWM values, show that our collector does not generate much additional memory for test cases that utilize small amounts of dynamic memory (fannkuch, mandelbrot, and pidigits). Of these fannkuch performs the worse for memory, but only by 61.44 KB of additional space as compared to the plain version. For mandelbrot and pidigits no dynamic memory is requested in the source file that we analyze and transform. Therefore, these tests provide an interesting case to evaluate how performance is affected by a relatively idle rbmm and rbmm+gc system. Potentially, the libraries that mandelbrot and pidigits make calls to might request dynamic memory; however, those libraries were never compiled with a region-aware compiler and are unaware of RBMM. Therefore, all (potential) memory requests in those libraries are all handled by Go’s existing collector. (This is the reason why we cannot fully disable the need for Go’s existing garbage collector.) In other tests (meteor-contest and binary-tree), where dynamic memory is more stressed, we find that the former performs comparable to the
unmodified test for rbmm and slightly worse for rbmm+gc. We see that the latter performs quite well when not running with our garbage collector, but uses significantly more memory with our collector. We believe this to be the result of our collector’s design; each region instance contains a list of zone header structures from which the region can allocate memory. This test generates numerous regions and we find that a majority of the zone headers are never used. Recall that each region allocates a zone header for every type in the program, which simplifies our design at the cost of additional runtime memory. The optimal choice would be to instantiate a region with zone headers for types that our analysis can establish at compile-time. When a type is allocated at runtime that does not have a corresponding zone in the region, a new zone should be allocated dynamically and added to the region’s list of zone headers. It would take quite a bit of extra work to implement this optimization and even longer to ensure reliability.

We find a nice memory improvement using rbmm over that of the unmodified version of the matmul benchmark. However, the rbmm+gc case also shows us that our garbage collector needs work as we could not run it to completion during testing. Once again, memory management is hard from programmers to get correct, and the same goes for memory management designers. matmul relies on slices, and our garbage collector’s slice handling has been riddled with bugs. Much time has been spent trying to smooth things out, with little to no avail.

As expected, our collector does utilize more pages than our rbmm system alone, as seen by the Inuse Pages metric. Recall that a region will need to allocate more pages during copy-collection for copying objects from from-space into to-space.

Table 5.4 shows the number of regions that our RBMM system creates for each benchmark. Fewer regions means that our system can run faster with less overhead of region creation and reclamation, but also means that our system is performing less work. However, even for the binary-tree case, which utilizes a relatively high region count, we still perform better,
with respect to time and space, than the base case. The main drawback is that our collector does seem to not benefit (reduce) the memory footprint of applications significantly, which is our collector’s primary goal. Regions can also adversely affect the HWM value for our tests. We find that our memory usage is high (larger than the unmodified programs) when we utilize our garbage collector. However, when we rely on Go’s garbage collector we find a comparable, or better, memory usage than the unmodified version.

Even though binary-tree and meteor-contest utilize many regions, we see that they are not compromising program performance. It is only when we add in our region-aware garbage collector do we see considerable negative performance. We believe the issue to be related to how our implementation utilizes region headers. Thus, the problem is not solely mishandling of the programs data, but also (if not nearly entirely) for handling bookkeeping data internal to our system. In addition, we have not optimized our collector, and thus it is still primitive at best.

Both of our modified systems perform comparable for execution time on all of the tests we measured. Since our time differences are often quite minor and the times so small, we cannot exclude the possibility of OS noise and overhead from other concurrently running processes on the test machine. As with the results seen in Chapter 3, our binary-tree performance did quite well over that of the unmodified version. However, it is also beneficial to know that the collector does not seem to pose significant time overhead for the low-memory tests.
Time is dependent on both application runtime and GC performance. Both GC systems, our garbage collector and Go’s existing garbage collector, are stop-the-world, and halt execution of the mutator to perform their GC duties. In addition, our RBMM systems additionally affect runtime performance due to our static analysis inserting region operations during compilation, which increases program size.

Table 5.3 shows the number of GCs that each test performed as well as the number of regions created. Note that we only ran this data collection once to obtain the GC values. One would expect the same collection counts on benchmarks which do not make any time-based or pseudo-random decisions. In fact, having benchmarks with repeatable output is desirable, as it allows for multiple identical independent runs of the same executable. However, we noticed that Go’s garbage collector had an interesting property, non-determinism. We can run a test with deterministic output (no apparent \texttt{rand()} calls or time-based decisions) numerous times and get different collection counts reported by the Go collector. We cannot guarantee what happens in external library calls, but we did not see any of the benchmarks making calls to the \texttt{random} or \texttt{time} Go packages. It turns out that the Go runtime system provided by our test GCC versions 4.7.2 and 4.6.3 enable memory profiling by default. We assume versions between the two aforementioned GCC versions also have the same property of permitting memory profiling by default. The Go memory profiler samples memory based on a random value. This sampling has an effect that can increase or decrease the number of time their collector executes. The number of collection counts are important as they allow us to decide whether or not our speedup is due to our GC frequency, RBMM efficiency, or possibly some combination of both. However, we report just the value from a single run from the default Go execution. We believe that such a number will still allow us to measure our speed up performance. The collection count values for our region-aware collector are only available on the tests that have our collector enabled \texttt{rbmm+gc}, all other cases have these values labeled as \texttt{NA}. 

156
We expected that our garbage collector would slow down the performance of our RBMM system. Surprisingly, both fannkuch and mandelbrot perform slightly better than rbmm, but only by 0.3 seconds in the case of fannkuch. We should note that the aforementioned tests only attempt to collect garbage two or one times respectively (as seen in Table 5.3). Further, the difference in times between rbmm and rbmm+gc is well within the window of noise that could be caused by OS overhead (e.g., process scheduling).

Our region-aware GC (rbmm+gc) did appear to slowdown runtime from that of the rbmm for binary-tree. This is what we expect, but the difference caused by our system’s 17 collection attempts was only 1.32 seconds. We also note that our rbmm system alone reduces Go’s collection from 202 to just 1. Thus, we witness a 3.97 fold speedup for rbmm and 2.7 fold speedup for rbmm+gc in comparison against the unmodified version. We note that a majority of the speedup in the latter is due to less GC.

pidigits is interesting as this test case triggers no GCs from our collector which suggests that this test makes a majority of its memory requests in non-region-aware libraries.

For matmul it seems that nearly all allocations are handled in the source file which means our RBMM system is fully utilized. Our rbmm system with Go’s GC does perform slightly better for space than the unmodified version. In this rbmm case Go’s collector was only utilized once which means that our RBMM system without GC, in cases where it can be fully utilized, can perform well.

We conclude that our base RBMM system without our garbage collector does not compromise runtime overhead. We cannot say the same for our system with our region aware garbage collector.
5.8 Summary

In this chapter we have described an augmentation to the design of our previous automatic memory management system combining the fast allocation and deallocation of memory under RBMM with the relatively low peak memory usage of a garbage collector. However, our results show that more work is needed to accomplish the goal of reducing the peak memory. As with our initial incarnation, our system still requires no programmer annotations, which we view as a benefit. To improve locality of reference, we use a copying collector, so we need to know the type of each item. Instead of attaching type tags to individual memory items, as done by Hallenberg, Elsman and Tofte [38], we attach them to pages, similar to the *Big Bag of Pages* approach to GC. Our solution is more closely related to Elsman’s later work [25], whereby he utilized the *Big Bag of Pages* idea for the Tofte and Talpin typed RBMM system. Our GC design can decide which region or regions to garbage collect based on runtime memory usage; however the implementation we developed collects from all regions all at once. It is easier to implement such a system where all regions are collected. The design described in this chapter can be modified to use compile-time region points-to information, to avoid tracing regions that cannot point to regions being collected. This can reduce GC times. In the future, our design can be extended based on the design of the Garbage-First collector [21]. The latter is a parallel and concurrent collector which can collect from a subset of regions. Unlike our solution, their design is not static; regions are based on temporal properties rather than object type.

We have also presented a design for garbage collecting unused elements in arrays. These can arise when slices and slices of slices are taken, old arrays and slices go out of use, and new ones continue to be used. Our design reclaims unneeded elements over two successive GCs, with each GC retaining the elements that were reachable at the time of the previous GC, and determining the elements needed during the current GC. This capability costs two bits per slice element and increases runtime overhead slightly, but
can potentially reclaim a substantial amount of additional memory. Once implemented, this feature can be switched off selectively in zones where it proves ineffective.
Chapter 6

RBMM in a Concurrent Environment

Scientific inquiry shouldn’t stop just because a reasonable explanation has apparently been found.

Neil deGrasse Tyson

Multi-core processing has become a common way of improving computational performance in lieu of single core systems meeting physical limitations. Parallelizing computations across multiple CPUs is a relatively simple way to achieve computational performance which has become the norm for even consumer grade hardware. It is not difficult to understand that if you give a powerful CPU to a research student they can find a way to maximize its performance relatively quickly. The same goes for the consumer world. In 1965, Gordon Moore established the notion that every two years the number of transistors on a integrated circuit doubles. This observation became known as “Moore’s Law,” and can more generally be reduced to the idea that computer systems will roughly double their CPU power every two years.

As physical limitations to CPU architectures are being approached, chip manufacturers are looking at other ways to continue providing consumers with more computation power. One solution is to merely provide more CPU execution cores for processing (multi-core systems). However, this leads to a different challenge: If a program is to achieve the best performance, then
programmers must write programs that take advantage of the increase in CPU parallelism.

Up to this point in this thesis, we have discussed RBMM for a sequential subset of the Go programming language. In this chapter we introduce a design that allows RBMM to work within a co-routine environment as provided by the Go programming language. To our knowledge this is a unique contribution that introduces RBMM into a language whose concurrency is designed around the paradigm of Hoare’s *Communicating Sequential Processes* [49].

### 6.1 Introduction

Often the terms *parallel* and *concurrent* are confused. The topic of concurrency versus parallelism is a seemingly popular topic within the Go community. Go developer Andrew Gerrand summarizes the differences as: “Concurrency is about dealing with lots of things at once. Parallelism is about doing lots of things at once” [33]. To clarify, a concurrent system is one that can execute multiple logical threads all at once. These threads are considered logical and can be of a quantity greater than the number of physical threads (system threads) and execution cores provided by the CPU. These thread executions may be interleaved in time (possibly on just a single CPU) and scheduled via the OS. In fact, a program can make use of threads and permit their application to operate as many smaller processes. This can enhance efficiency and allow a program to perform multiple tasks all seemingly at the same time. If a programmer wants to squeeze the most performance out of their application, they should consider ways to thread their application in order to take advantage of such concurrency. In contrast, parallel programming makes use of multiple execution cores or system threads so that the threads of a single program can execute simultaneously (in parallel). For the remainder of this chapter we will interchangeably refer to the threading of a program and allowing it to execute in a parallel context as *concurrent*
or parallel programming. The differences in meaning are not necessary to explain our modifications.

While hardware seems to be increasing in performance, easy solutions to concurrent programming are still primitive. Programmers must be aware of the dangers of deadlock and race-conditions, and try to avoid them as best as possible, using whatever programming language features are available. Deadlock occurs when multiple threads are trying to access a shared (critical) piece of data, and each must wait on the progress of the other thread(s). In such a case, the program will appear to have paused. Race-conditions occur when the access to a piece of shared data is dependent on a particular sequence of events. A programmer must prevent one thread from reading or writing data that another thread might be manipulating at the same exact time, otherwise one of the threads (or possibly both) will have an incorrect picture of the program’s state. We call such invalid data access a data access contingency.

As mentioned, no simple solutions have become common to avoid such situations. Mostly, a programmer must employ complicated locking constructs to restrict access to a critical piece of data (critical section) permitting only one thread to read/write it at a given time. This is hard! Rather than being able to focus on solving their primary computational problem, programmers may also have to manage resources (e.g., memory management), all while trying to accomplish these tasks in concurrent contexts.

### 6.2 Go’s Approach

The Go programming language aims to reduce the complexity of concurrent programming by encouraging a practice of sharing memory by communication [32]. To do this, Go relies on co-routines and channels as introduced in Tony Hoare’s 1978 paper, *Communicating Sequential Processes* [49]. Such a system eliminates the need for a programmer to explicitly lock data shared among threads. Instead, a program can be written using channels and co-
routines (in Go terminology these are called go-routines) that permit atomic access to data. Instead of having multiple threads look at a single piece of data (potentially at the same time), Go encourages programmers to pass data between go-routines using channels. A go-routine can listen for data on a channel, and will block if that data is not available. Once a routine executing on a thread gains access to the data, it can manipulate it. If the program was designed properly, the need to lock the data is handled implicitly by the blocking that occurs while the routine is waiting to receive data from the channel. Thus, it is easy for a Go programmer to write a parallel program and share data atomically to avoid some of the parallel programming pitfalls.

6.3 Design

While Go programmers might have an easy means of applying concurrent models to their software, our implementation of parallel-safe RBMM must still rely on more traditional forms of handling concurrent data access. In fact, our RBMM-aware Go runtime system is all written in C, and we rely on traditional (complicated) constructs to prevent data access contingencies.

In this section we present a design for managing regions that can be used in concurrent Go programs. Any function in Go can execute concurrently by simply prefixing the function call with the keyword `go`. The new function invocation will then execute in a new independently-scheduled thread, which will terminate when the call returns.\footnote{The Go language and runtime system is designed to handle thread creation cheaply. Instead of spawning off multiple system-threads, which can be resource expensive, the language permits go-routines to execute (interleaved) on a single system thread.}

Since the new thread can execute in parallel with its parent, operations on any regions passed from the parent thread to the new thread will need synchronization. To support go-routines we mark regions passed in such calls, and generate calls to modified versions of the region creation, allocation, and removal operations.
To better understand our design we provide the following additions, in Figure 6.1, to the analysis constraints originally described in Section 3.6. Both send and recv require channel variables. Since channels are allocated

\[ S[v_1 = \text{recv from } v_2] \rho = (R(v_1) = R(v_2)) \]
\[ S[\text{send } v_1 \text{on } v_2] \rho = (R(v_1) = R(v_2)) \]
\[ S[\text{go } f(v_1 \ldots v_n)] \rho = \theta(\pi_{f_1 \ldots f_n}(\rho(f))) \]
where \( \theta = \{f_1 \mapsto v_1, \ldots, f_n \mapsto v_n\} \)

Figure 6.1: Added rules for handling Go’s concurrency primitives.

with new, they are associated with regions. Go’s go statements take a single function call as input. Our semantics maps the regions of the actual parameters of the caller to the formal parameters of the callee. This mapping generates a set of regions for all of the input arguments. Notice that we ignore return values. Any function can be made parallel via the go keyword, even if it returns values. However, Go will not permit the assignment of a return value from a go-routine, thus syntactically enforcing the programmer to make use of channels. In Section 6.3.3 we justify the requirement for having messages being sent and received to share the same region as the channel they use.

### 6.3.1 How a Go-routine Executes

To understand the following sections, it is necessary to understand what happens at both compile and runtimes when a go-routine is created. The compiler will first replace the go-routine call with a special helper-function that is responsible for creating a new thread and then calling the function that is to be made concurrent. This helper function takes a closure and a
set of arguments. The closure is that of the original function that is to be made concurrent, and the set of arguments are the original arguments passed to the function. At runtime, the helper function is called, a new thread is created, and a wrapper function is called. The wrapper executes on the go-routine and only passes the proper arguments to the closure, executes the closure, and the wrapper exits. Once the wrapper returns, the go-routine can be thought of as being terminated.

6.3.2 Concurrency for Region Operations

As with our semantics, we must also reconsider the meanings of our region operations, which we originally described in Section 3.5. The important consideration is how to prevent any data contingencies when accessing the data that these operations manipulate (region metadata). Such accesses must be atomic. If not, a data access contingency can arise.

Protection Counters and Concurrency

Our protection counters have been made thread safe via the use of atomic operations. This ensures that no two threads can manipulate a single counter at the same time.

Our RBMM system must take care during region removal. It must prevent a thread from prematurely removing a region that might be accessed via other regions (and might also be shared by a multitude of threads). In Section 3.8 we introduced the concept of a protection counter which prevented a function from prematurely removing a region that was needed by a calling function. This idea needs refinement given a concurrent context. Consider the case where one go-routine has completed its use of a region, and then decrements the region’s protection counter and attempts to remove that region. Potentially another go-routine might still be using that region, and if it suddenly gets removed, then the resulting program is incorrect (and can crash). To prevent such cases, our protection counter takes on a slightly
different meaning in the context of go-routines: A region’s counter reflects the number of caller functions and go-routines that need the region alive. Consider another example:

```go
func main() {
    reg := CreateRegion()
    val := AllocFromRegion(reg, sizeof(int))
    *val = 42

    // More code...
    go processSomeData(val)

    // More code...
}
```

In the protection counting scheme used so far, we would increment the counter just prior to calling `processSomeData` and decrement it just after that call. Go-routine calls spawn off a new lightweight thread of execution and the current thread will proceed to the next statement irrespective of how long the go-routine’d function takes to execute. If we were following our old scheme we would now remove the region associated to `val`, oblivious to the fact that the go-routine, which concurrently executes `processSomeData`, might access `val`. This could result in a program crash. To ensure that our counters are thread

To prevent this, we increment the protection counter, as usual, just prior to making the go-routine call. The adjustment we introduce here is that we decrement the counter once the go-routine completes. To understand the reasoning behind this adjustment, it is necessary to understand how a go-routine operates at the low-level, as explained in Section 6.3.1. Our solution is to insert the decrement and region removal operations in the `wrapper` that is responsible for executing the call at runtime.
Region Header and Concurrency

We add a lock variable to our region header data structure. This variable is used by our runtime system to prevent multiple threads from removing or allocating from a region simultaneously. It is important to note that we lock just the region and not the entire allocator, allowing non-locked regions to be allocated from or removed without care of what is happening to other regions.

CreateRegion() and Concurrency

For region creation operations, our parallel modification allocates space for, and initializes, the additional lock field in the region header.

AllocFromRegion() and Concurrency

For region allocation operations, our modification turns the usual code of the operation into a critical section that is protected by the lock field in the region header. This extra synchronization can be optimized away on allocation operations in the main thread before the first go-routine call involving the region is executed.

RemoveRegion() and Concurrency

For region removal operations, our modification operates, under mutual exclusion, on the lock field in the region header. When the region is mentioned as an argument in a go-routine call, we increment its protection counter. This signifies that another thread has access to the region and that the region should not be removed until no other threads have access to it. When no other threads need to access the region’s contents, the protection counter will have a value of zero, signifying that the next region removal operation can safely reclaim the region’s memory.

Just before a thread executes an operation to remove the region, at the point where it has no further references to the region, we decrement
its protection counter. This decrement occurs in the wrapper discussed in Section 6.3.1. If the region’s counter is still positive, some other threads must still be using the region, or a variable in the region is still needed later in the function’s execution, so the removal operation will not be able to reclaim the region’s memory. This runtime test is necessary because, while a static analysis can figure out which program point in the body of each thread makes the last reference to a region in that thread, the question of which of these per-thread last references will actually be executed last at runtime may depend not only on the input to the program but also on the effects of thread scheduling, and thus in general cannot be decided statically.

### 6.3.3 Transformation

The overall transformation is shown in Figure 6.2. Note that the function \( f \) has been replaced in the transformed version with a wrapper function, \( f' \) as we discussed in 6.3.1. Our wrapper also ignores the return value of \( f \) if it were to have one. Since a call to a go-routine completes immediately, and the execution of the go-routine might be delayed due to low-level scheduling reasons, there is no return value that the caller can use. Instead, channels should be used to communicate data between go-routines (a singly-threaded application is a single go-routine). Like \( \text{main} \), when \( f' \) exits, its thread will not have any remaining references to the regions it handles, but unlike \( \text{main} \), it gets some regions from its parent thread, and does not have to create them all by itself.

Note that the \texttt{IncrProtection} operations must be performed in the parent thread; if they were in the child thread in \( f' \), the parent thread could delete a region before the child thread gets a chance to perform the increment that would prevent that.

We can optimize the above code in some cases. For example, we can guarantee that some per-thread last references cannot be the last reference globally. If two threads communicate using an unbuffered channel, meaning that the writing thread will block until the reading thread is ready to read,
and if the last reference to a region in the reading thread is before the read while the last reference to that region in the writing thread is after the write, then we know that the last reference to the region in the reading thread cannot be the overall last reference to the region. In that case, we can optimize away the call to RemoveRegion after the call to DecrProtection in the reading thread.

When a thread \( t_1 \) sends a message to another thread \( t_2 \), with a statement such as send \( v_1 \) on \( v_2 \), the code executed by \( t_1 \) effectively decides what region supplies the memory for the message: it will be \( R(v_1) \). When \( t_2 \) receives the message, it will do so with a statement such as \( v_3 = \text{recv from } v_4 \). After this statement, \( t_2 \) will believe the message to be in region \( R(v_3) \). We need this to be the same as \( R(v_1) \), since otherwise the two threads will disagree about when the region of the message can be reclaimed. We ensure this by imposing this chain of equalities: \( R(v_1) = R(v_2) = R(v_4) = R(v_3) \). The first equality is from the analysis rule for send statements; the third is from the rule for recv statements; and the second follows from the fact that for the message to be transmitted, \( v_4 \) must refer to the same channel, and thus the same region, as \( v_2 \).

There are two ways that two threads can communicate. One way is for both to be given a reference to the same channel by a common ancestor.
(which may be one of the threads themselves). In this case, a variable representing the channel will be an argument in a go-routine call, and therefore after our transformations, the region of that channel will be passed along with it. The other way is for one or both of the threads to receive the channel in a message. The design presented in Chapter 3 stores all parts of a data structure in the same region. In Chapter 5 we relaxed this constraint, permitting a single data structure to have multiple regions associated to itself and its members. Our analysis must place channels and the data they refer to in the same region. Likewise, our analysis must also place any message used for passing channels in the same region as that channel. This implies that (a) a channel in a message is stored in the same region as the message, while the rule for `send` operations says that (b) a message is stored in the same region as the channel it is sent through. Together (a) and (b) imply that, if a channel $c_2$ is sent in a message on channel $c_1$, then $R(c_1) = R(c_2)$. This means that even if $t_1$ and $t_2$ communicate on channels sent in messages, those channels use only regions whose identities are passed between threads at go-routine calls.

Our system of equating the regions of messages and channels allows the region of a message to be reclaimed while the message is in a channel only if the channel itself is being reclaimed. This can happen if, after a message is sent on a channel, all references to the channel become dead. If that happens, no thread can ever receive the message, so recovering its memory is safe.

### 6.3.4 Producer and Consumer Example

The example in Figure 6.3 illustrates the design presented in this chapter. We choose to use a common concurrency pattern, a producer and consumer model. In this example the `main` function acts as the consumer and is fed *wookies* from a farm of wookie producers. Each producer is merely a function called `producer` and executes in a separate go-routine thread. These producers generate wookies for the consumer to manage. The example
generates 10 producers and will terminate once the consumer has devoured a mass of 1000.0 WookieTons of wookie.

The producers and consumer communicate via a shared (unbuffered) channel, \( ch \). The main thread of execution creates a shared channel and passes it to all of the producers. Each producer will continuously generate a \emph{Wookie} object, populate its fields, and pass the new wookie back to the consumer. The producers only stop once the program terminates, which is a decision made by the consumer.

After our RBMM transformation, the code will look similar to that presented in Figure 6.4. Our transformation establishes a region that the wookie instances are allocated from. Since the consumer will need the wookie instances to consume, it must also create the region for the wookies it will receive. Since the consumer is communicating with the producers via a shared channel, and the wookie instances are sent down this channel back to the consumer, our analysis will unify the regions for wookies and the channel. This unification results in a single shared region consisting of the channel and the wookies that will be sent down it.

Our transformation also generates a new function, \emph{producer\_wrapper}. This new function is the result of analyzing the \emph{go} statement during compilation. The \emph{go} statement will create a separate thread of communication by copying the \emph{producer\_wrapper}, and the original arguments to \emph{producer}, onto a new thread of execution. This wrapper is responsible for calling the original function (\emph{producer}) with the proper arguments and then decrementing the region as appropriate.

When this program is executed, our system will try to remove the region in the wrapper, but since the consumer incremented the region’s counter, this removal will never succeed. Instead the program will exit before all go routines complete, due to meeting the desired consumed wookie mass. In this case the consumer will never remove the region. The region’s protection counter will always be greater than zero due to the constantly running producers utilizing the channel that belongs to the region. This is correct
package main
import "math/rand"

type Wookie struct {producer_id, wookie_id int; weight float64}

func producer(producer_id int, ch chan *Wookie) {
    counter := 0
    for {
        counter++
        w := new(Wookie)
        w.producer_id = producer_id
        w.wookie_id = counter
        w.weight = rand.Float64() + 42.0
        ch <- w
    }
}

func main() {
    ch := make(chan *Wookie)
    for i:=0; i<10; i++ {
        go producer(i, ch)
    }
    mass := 0.0
    for {
        wookie := <-ch
        mass += wookie.weight
        println("Wookie", wookie.wookie_id,
            "from", wookie.producer_id,
            "weighs", wookie.weight, " WookieTons")
        if mass > 1000.0 {
            break
        }
    }
    println("Consumed", mass, "WookieTons of food")
}

Figure 6.3: Producer/consumer example

since our system should not remove a region for a channel when its data is needed later.
func producer_wrapper(producer_id int, ch chan *Wookie, reg *Region) {
    producer(producer_id, ch, reg)
    DecrProtection(reg)
    RemoveRegion(reg)
}

func producer(producer_id int, ch chan *Wookie, reg *Region) {
    counter := 0
    for {
        counter++
        w := AllocFromRegion(reg, sizeof(Wookie))
        w.producer_id = producer_id
        w.wookie_id = counter
        w.weight = rand.Float64() + 42.0
        send w on ch
    }
}

func main() {
    reg1 := CreateRegion()
    ch := AllocFromRegion(reg1, sizeof(chan *Wookie))
    for i:=0; i<10; i++ {
        IncrProtection(reg1)
        go producer_wrapper(i, ch, reg1)
    }
    mass := 0.0
    for {
        wookie := recv from ch
        mass += wookie.weight
        if mass > 1000.0 {
            break
        }
    }
    RemoveRegion(reg1)
}

Figure 6.4: Producer/consumer example with region annotations
6.4 Region-Aware Garbage Collection

Chapter 5 discussed our region-aware garbage collector. The solution discussed in this current chapter should be able to operate using our garbage collector so long as the threads that might use or manipulate a region being collected are paused for the duration of GC. Threads that cannot access any regions being collected may be allowed to continue during GC. Our implemented system handling go-routines does not use our region-aware collector due to its buggy and inefficient status.

6.5 Evaluation

To test the effectiveness of our RBMM transformations for supporting Go’s concurrency capabilities, we have modified our GCC-plugin to transform code as discussed in this chapter. We look at both time and space metrics to gauge the performance of our modifications.

6.5.1 Methodology

As with our previous experiments, we rely primarily on benchmarks from Debian’s “Computer Language Benchmarks Game,” provided by the gccgo compiler. The Go team has modified a subset of these tests to support concurrency via go-routines, and it is these tests we specifically focus the following evaluation on.

Some of these tests we have not looked at before: k-nucleotide-parallel, regex-dna-parallel, and threadingring. The first two take an input file representing a DNA sequence and perform operations such as counting DNA sequences or looking for a particular sequence match. threadingring spawns numerous go-routines which share data via communication between “adjacent” threads in a circular/ring of threads. The shared data that is communicated and manipulated is merely an integer that is decremented, thus an inexpensive operation.
We validated the output from our modification \textit{(rbmm)} and compared that data to the output of the unmodified versions \textit{(plain)} to ensure our results match that of the unmodified test. During test execution, we disabled the benchmark’s output. This helps eliminate some OS overhead which has little to do with the test’s actual performance. Each benchmark was run 10 times to obtain an average high water mark \textit{(HWM)} and \textit{Elapsed Time} value. We also disabled our garbage collector, as we are less confident of its implementation. Instead, we relied on Go’s existing garbage collector as we did in Chapter 3.

The region count values are only available for the modified tests, and since these numbers are deterministic, they were gathered once and not sampled 10 times. \texttt{threading} intentionally terminates early by calling \texttt{os.Exit()}. Our statistical counters are only displayed upon return from \texttt{main}. For the \texttt{threading} test, we had to run the application in the GNU debugger (GDB) and set breakpoint on the \texttt{os.Exit} call. This breakpoint allows us to look at our statistical counter value representing the number of regions created. Traditionally, in sequentially executing Go programs, we would validate our system during development by looking at our statistical counters. The number of regions created and removed should have a delta of 1 by the end of execution. The 1 region difference represents the global region, which we never remove. For tests using go-routines, we cannot rely on this delta. For instance, a program can terminate before all go-routines complete. Often, the parent go-routine and its child go-routine(s) are naturally synchronized via channels. The wrapper that wraps the function call declared in the \texttt{go} statement will try to remove regions after the child has responded to its parent via channel communication. If the parent’s execution is quick enough, the program might terminate before the child go-routine and wrapper complete. This can result in a delta greater than 1 between the number of regions created and removed. Therefore, we cannot rely on this statistic for an accurate measurement of our system’s correctness. However, the number of regions created does reflect that our system is doing something...
and this value can help us gauge our performance and potential overhead that our RBMM system might impose during program execution.

Our experiments were conducted on the same hardware as our previous two evaluations (see Chapter 3. That system is now running Ubuntu 13.04. We used GCC version 4.7.2 (which supports the Go version 1.0 specification) to compile both our modifications (our plug-in and runtime) and tests. The GCC Go libraries used during runtime were provided by the same GCC 4.7.2 and are from Go version 1.01. Some of our modified runtime is based on code from GCC’s libgo version 4.7.1 and 4.7.0.

6.5.2 Results

Table 6.1 shows our performance results from this experiment.

Our timing results show comparable performance with that of the unmodified tests. Even though we do have a bit of region management overhead, we still perform well in most of these cases, and only worse on one test, spectral-norm-parallel. However, our tests were short-lived and might reflect some OS performance overhead that is not directly related to our tests, such as the scheduling of other processes. We cannot say conclusively that we are better or worse than the unmodified version with respect to time, but we appear to be comparable for programs with short execution times.

Our memory performance is comparable as well. In most cases we increased the memory footprint of the process. We witnessed a 20.48 KB increase in the high water mark (HWM) for fannkuch-parallel, and 409.60 KB for regex-dna-parallel. We also had a negative impact on memory, to the order of 4.01 MB for k-nucleotide and 2.00 MB for threading. An increase in memory is not unheard of for an RBMM system, and this can be the result of the region-bloat problem where the region keeps unused data around to the program point where our static analysis found a place to safely reclaim the region. We note that adding our go-routine-capable RBMM system adds on average 83 KB to the binary file size of the bench-
<table>
<thead>
<tr>
<th>Test</th>
<th>HWM (MB)</th>
<th>Elapsed (seconds)</th>
<th>Regions</th>
<th>Executable Size (KB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>fannkuch-parallel (plain)</td>
<td>8.55</td>
<td>4.32</td>
<td>NA</td>
<td>84</td>
</tr>
<tr>
<td>fannkuch-parallel (rbmm)</td>
<td>8.57</td>
<td>4.28</td>
<td>6</td>
<td>168</td>
</tr>
<tr>
<td>k-nucleotide-parallel (plain)</td>
<td>39.02</td>
<td>1.23</td>
<td>NA</td>
<td>87</td>
</tr>
<tr>
<td>k-nucleotide-parallel (rbmm)</td>
<td>43.03</td>
<td>1.29</td>
<td>15</td>
<td>170</td>
</tr>
<tr>
<td>regex-dna-parallel (plain)</td>
<td>25.54</td>
<td>2.46</td>
<td>NA</td>
<td>76</td>
</tr>
<tr>
<td>regex-dna-parallel (rbmm)</td>
<td>25.94</td>
<td>2.47</td>
<td>11</td>
<td>157</td>
</tr>
<tr>
<td>spectral-norm-parallel (plain)</td>
<td>8.75</td>
<td>0.84</td>
<td>NA</td>
<td>73</td>
</tr>
<tr>
<td>spectral-norm-parallel (rbmm)</td>
<td>8.61</td>
<td>0.80</td>
<td>182</td>
<td>159</td>
</tr>
<tr>
<td>threading (plain)</td>
<td>13.69</td>
<td>6.18</td>
<td>NA</td>
<td>65</td>
</tr>
<tr>
<td>threading (rbmm)</td>
<td>15.69</td>
<td>6.15</td>
<td>506</td>
<td>146</td>
</tr>
</tbody>
</table>

Table 6.1: Go-routine performance with and without RBMM

Our main concern in these tests is \texttt{k-nucleotide-parallel}'s relatively large HWM. Recall that the HWM value reflects max resident set size used by the process. It seems that this program in particular does not reuse many regions, thus will increase the HWM more so than if the regions were more frequently reused. Since our allocator never releases memory back to the OS, it makes sense that we might have a larger HWM than the unmodified test.

Ultimately, we find that our system does not change running times significantly, even though we have added concurrent safety measures (locks and compare and swap concepts) to prevent our runtime from having data access contingencies. We also note that we have not spent much time optimizing our modified system.

### 6.6 Summary

In this chapter we introduced a design to handle concurrency for RBMM in an imperative language with CSP-styled parallelism. Our design is based around the concept of the protection counter which we believe to be a unique
contribution of our research. This counter’s semantics has been updated to represent the number of functions that need the region at a later time as well as the number of concurrent processes that need a specific region alive. Removal of such a region cannot occur until the counter has reached a value of zero, signifying that no threads need the data allocated from that region anymore. As with the rest of our work, our focus has been on the Go programming language, but with some modifications the design can be modified to fit other languages, especially those with a similar CSP-styled concurrency system.
Automatic memory management is an established research area of computer science that has its start in 1959 with the implementation of a garbage collector for the Lisp programming language [60]. It is well understood that handling the management of memory is not a trivial task for a programmer, especially when the task is orthogonal to the problem that is to be solved. A language’s runtime system and compiler can maintain a higher degree of safety if the (error-prone) human programmer is relieved of the burden of memory management. This chapter looks at various implementations of automatic memory management and related concepts which further enhance the understanding of this thesis: the combining of RBMM and GC within an existing language.

7.1 Garbage Collection

Early in computing history it was realized that permitting the system to manage memory automatically eases the job of the programmer. In 1959, John McCarthy and his team at MIT’s Research Laboratory of Electronics introduced the concept of a garbage collector into the Lisp programming language. The original implementation was a mark-sweep collector, which works in two phases. When the system runs out of free memory, the collector begins its marking phase, in which it scans memory, starting at the base...
system registers, and tracing pointers. All visited objects have their sign-bit set, which signifies that they are reachable via one of the base system registers. In the second phase, the collector sweeps the entire memory space, returning any unused (unmarked) memory back onto a freelist, which can be used for later allocations. Since GC is “entirely automatic” it is “more convenient for the programmer” to let the system “keep track of and erase unwanted lists.” The authors found that this system can be time-expensive, as “the reclamation process requires several seconds to execute” [60].

With more and more languages adopting GC as a means of automatic memory management, research into improving the performance of GC has greatly increased. In order to improve collection speeds, it is necessary to get a better understanding about the properties of allocated objects.

The weak generational hypothesis of GC is an observation that the most frequently collected objects happen to be objects that are the most recently allocated (the youngest objects)[80]. One method to reduce the overhead of GC is to scan only a well-chosen subset of the entire memory space for collection. Generational garbage collectors exploit this observation. Ungar’s generational scanner [80], for Berkeley Smalltalk, showed a notable improvement over past GC techniques, such as reference counting (33% improvement over deferred reference counting). Ungar observed that the copying of surviving objects to a newer generation has lower overhead than scanning the dead/unreachable objects.

Lieberman and Hewitt introduced a real-time GC algorithm for partitioning the memory space into regions based on object lifetimes [58]. Regions are arranged in generations and are scanned at a frequency based on the lifetime of objects they contain. Generations that contain regions of younger objects are scanned more frequently than generations containing regions of older objects. Any reachable objects in a region that is being collected (scavenged) are moved into a newer region. The memory for the scavenged region can then be reclaimed. Lieberman and Hewitt’s algorithm makes use of the fact that in certain systems (e.g., Lisp) pointers more com-
monly point backwards in time. In other words, objects are composed of pointers that were created earlier during program execution. The authors note that shorter-lived objects account for a higher proportion of memory space, therefore it is useful to scan them more frequently for garbage than older objects.

GC performance can be improved by offloading some analysis to compile-time. This static analysis can be used to reduce the cost of executing collection cycles at runtime. As early as 1977, Barth investigated the use of static analysis to benefit GC [6]. Barth’s approach groups allocations of objects into classes. Reference counting is then performed on the entire class and not per-object, which reduces the overhead of per-object counting. This concept is similar to our region-aware garbage collector as described in Chapter 5. Our regions are composed of zones, where there is one zone per data type in the region. In contrast to Barth’s work, our regions, as a whole, are reference counted (protection counted). This count is not per object or per data type in the region as Barth’s was. Instead, our counter reflects the fact that the region will be needed later and should not be removed by a callee function.

In 1988, Ruggieri and Murtagh introduced lifetime analysis [67]. This static analysis technique can obtain an upper-bound on how long any particular object lives. With this knowledge, an object which has an approximately bounded lifetime can be allocated at function entry and deallocated at function exit, reducing the need for GC and the associated overhead. Our region inference analysis is based on object lifetime. A region will have an upper-bound on its lifetime based on the life of the item in the region which lives the longest. However, we do not try to remove a region at function exit. Instead, we aim to reduce memory pressure and remove a region as soon as possible.

Barry Hayes showed that the weak generational hypothesis holds; young objects have short lifetimes [42]. However, he found that waiting longer to perform reclamation is not useful. His empirical studies suggested that
while lifetime is a reliable predictor of reclaimability for young objects, this is less so for older objects. Instead, as objects age into a particular generation (surviving multiple collections) another measure of collection is more attractive. Scanning a potentially mature and heavily populated older-generation is “unattractive” and can be a waste of computation time. Hayes’ system makes use of his observation that clusters of objects that are allocated at the same time often have a similar lifetime. His system marks an object of such a cluster as being a “key.” When that key object becomes unreachable, then the cluster should be scanned. This method reduces the amount of time spent collecting generations of older objects.

Hicks used a static lifetime analysis to verify the correctness of object deallocation [46]. This information can be used to insert deallocation calls into the program at compile-time. Hicks found that his implementation could deallocate 80-100% of the storage space allocated by the test programs he measured. The result applies to certain classes of programs and is not representative of all programs. Nevertheless, this finding strengthens the motivation for combining static analysis with a runtime garbage collector. This is similar to RBMM whereby a static analysis decides at what program points a region of objects can be reclaimed.

Aside from lifetime, object connectivity is another property of dynamically allocated data that can be used to optimize GC. In 2003 Hirzel et al. introduced a garbage collector that is based on an object points-to relationship. This collector is based on information from their earlier finding: objects that are connected tend to die together [48]. The authors found that partitioning the memory space based on object connectivity reduces the need for the garbage collector to scan the entire program’s memory space. Their results demonstrate a benefit over generational collectors with respect to mutator pause times and memory space utilization [47].

Khedker et al. demonstrated that a static analysis can be used to benefit time and space properties of GC [52]. The authors performed heap reference analysis to generate “access graphs.” These graphs are constructed from
“access paths” deduced statically at compile-time, and form an abstracted view of the heap. Such information can be used to better understand the lifetime and connectivity of objects within the heap. Unlike previous works, the authors looked not just at allocation sites, but at every program point which contains an object reference. By combining this information with dataflow information, the authors built an abstract picture of the heap. This information can be used to nullify objects that the compiler knows will not be used, which in turn may allow the garbage collector to collect referenced objects earlier. The authors found this reduced heap size and led to the system spending less time garbage collecting. They also found that less data had to be copied during collections.

Controlling the size of the heap can be used to reduce the amount of time spent in GC. Arjom and Li showed this to be the case for their threshold algorithm [4]. These thresholds act as a dynamically changing limit of the heap size. A GC occurs once this threshold is met. If the collection reclaims a certain, pre-defined, amount of memory, then the threshold will remain. Otherwise, the threshold will be adjusted until the amount of reclaimed memory meets it. This approach was designed to reduce paging by increasing the GC frequency when the heap gets to a particular size, instead of allowing the heap to continually grow. The authors showed that their heap-threshold improved runtimes, over that of the Boehm-Demers-Weiser conservative mark-sweep collector [8], on a series of Java programs.

Understanding the connectivity and liveness properties of heap data is not the only means for creating more efficient garbage collectors. Other design choices can heavily influence the amount of time spent in recovering unused resources. For instance, the interaction between the mutator and garbage collector threads can be designed in such a way that system pause times are reduced. It is possible to create a garbage collector that exploits the parallelism of a system, permitting the collector to operate concurrently with the mutator threads. In addition, garbage collectors can be made parallel, whereby multiple collection threads execute simultaneously during
Halstead showed that Concurrent Multilisp, a concurrent implementation of Lisp, can make use of a multiprocessor system to perform parallel copying GC. This implementation places heap memory spaces on each processor (semispaces), from which objects can be allocated. This eliminates contention between threads. The only point of synchronization occurs when the copying phase has completed and the memory spaces are swapped [39]. The collector is based on Baker’s copying collector [5], which divides the memory space into semispaces (sometimes called from-space and to-space) [27].

Baker’s algorithm works by dividing the memory into two semispaces and then coloring the objects in those spaces: white objects are those in the from-space, black objects are those that have been traced and copied, and grey objects are those that have been copied to to-space but have not been traced [5]. Baker mentions that, even though his copying collector compacts free-space and helps to reduce memory fragmentation, it can require excessive memory for some programs. This is an inherent property of semispace copying collectors; they require at least enough memory to support worse case copying. This case happens when all allocated objects survive a GC cycle and must be copied from from-space to to-space. A copying collector is an ideal choice for our region-aware garbage collector. The purpose of our collector is to reclaim memory from objects within a region. Since we want to reduce region size as much as possible, and remove unreachable objects (possibly in the middle of a region), the compaction that a copying collector provides is an ideal choice to accompany our RBMM design.

In 1993, Doligez and Leroy implemented a parallel GC for Concurrent Caml Light (a derivative of ML) [23]. In their system, each thread has its own heap. The heap on one thread can be undergoing GC while the other mutator threads are executing. Their system is a generational collector, whereby the younger generation is collected via an asynchronous copying algorithm and the older generation is collected via a concurrent mark-sweep...
algorithm. The young generation exists per thread and the older generation is accessible by all threads. As is common in generational collectors, all newly allocated objects are produced from the heap associated to the young generation. In this system, pointers are restricted to only point from the private (young generation) heaps into the shared (older generation) heap.

The Garbage-First garbage collector is a concurrent and parallel collector that can meet a soft real-time goal for Java [21]. This collector operates on regions of allocations. Object marking occurs in multiple phases, some of which operate concurrently with the mutator threads and others which require the mutator threads to be paused. The concurrent marking is the result of using a snapshot of the program state. This data are used to guide the marking phase. The actual collection (evacuation) occurs in parallel with the mutator threads paused. Unlike our RBMM + GC design, which generates regions based on object connectivity, the regions of the Garbage-First collector are temporal based. Their design permits a subset of regions to be collected. The name Garbage-First is in reference to what regions the collector chooses to collect from.

7.2 Region-Based Memory Management

While automatic memory management relieves the programmer of the burden of manually managing memory, it also can be a computationally expensive process. As we have seen, understanding how the heap is organized can provide clues that can be used to manage memory more efficiently. Even understanding the use of manual memory managers can provide additional insight to benefit automatic memory management. Berger, Zorn, and McKinley provided a thorough investigation of manual memory management, looking at both traditional and custom general purpose memory allocators, with some of the custom allocators being based on regions [7]. They also presented a new system, called a reap allocator, which acts as a combination of both general purpose and region allocators. A reap be-
gins life as a region. When a request to free an item from the region occurs, then the reclaimed memory from that item becomes memory for the region’s freelist. Once this freelist is fully utilized by the region, the region continues allocating from its end. Berger, Zorn, and McKinley show that on both memory consumption and execution time, the custom region allocators beat the Doug Lea allocator on three of five benchmarks, and the reap allocator on four of five benchmarks. These results are encouraging, even though the general purpose and reap allocators can recover individual objects and the custom region-based allocators cannot. On the other hand, automatic RBMM systems, such as the one presented in this thesis, may not be able to replicate the performance of these manually tuned region allocators.

A vast amount of RBMM research has been conducted by Tofte, Talpin, Hallenberg, Birkedal, and Elsman with the practical aim of improving the memory system of the Standard ML programming language [77]. The initial motivation for creating a RBMM system was to reduce memory footprint and present predictable runtimes for programs. This is in contrast to the garbage collector already provided by the language. In 1992, Talpin and Jouvelot presented a static analysis for inferring sets of referenced data (regions) in an extension of Core ML. They mentioned that such a system can be used for managing memory [75]. Soon after this research was published, Tofte and Talpin introduced RBMM for Standard ML [78]. Their seminal idea was to allocate data, based on lifetimes, into stacks of regions. They found that the maximum resident memory size tended to be better with the RBMM approach, while GC was often faster. The authors later proved the soundness of their method [79]. In 1998, Tofte and Birkedal published their region inference algorithm for ML and proved its soundness. The authors mentioned that, in certain cases, if a region has pointers into it, the region can still be removed if the system can prove statically that the pointers are never live at the point of region reclamation. In contrast, a garbage collector that traces references must conservatively assume that all reachable objects are live [76]. Our RBMM system does not impose a stack approach to
regions.

While RBMM was popularized through its application to ML, roots can also be found in work on imperative programming languages. In 1990 David Hanson published an article in Software Practice and Experience about his arena memory management system [41]. This system is for C and groups allocations based on object lifetimes. An object’s lifetime specifies which arena its memory can be allocated from. When an arena is deallocated, all objects in that arena are reclaimed all at once, therefore attaining the fast reclamation that RBMM systems provide. Hanson found that his system was faster than a quick-fit heap-based allocator and is less than double the speed of a stack-based allocator. Stack allocation introduces nearly a zero runtime overhead, and can be thought of as a best case (in speed) for memory management.

Aiken, Fähndrich and Levien observed that a stackless RBMM system can use less memory than Tofte and Talpin’s stack-based RBMM system [3]. The authors liberated region lifetimes from having to coincide with lexical scope, that is, from the stack discipline. Instead, their constraint-based static analysis transforms the input program by inserting region creation operations as late as possible and region reclamation operations as soon as possible.

Henglein, Makholm and Niss also explored implementing an RBMM system for ML based on reference counting [44]. The latter choice permits faster region reclamation than the stack of regions approach.

In 2000, Makholm introduced a Tofte-Talpin inspired RBMM system to a subset of the Prolog programming language [59]. In this study, Makholm found that RBMM is a reasonable alternative to GC on the benchmarks he measured. On three of five benchmarks, he found that RBMM is faster than, or equal to, the time overhead of a copying garbage collector. In two cases he found that his RBMM system caused the benchmarks to run 5-10% slower than GC. He suggested that this was the result of RBMM performing operations that were never used.
Hallenberg, Elsman and Tofte extended a stack-based RBMM system with a copying garbage collector using Cheney’s algorithm [38]. Unlike our approach, their GC requires adding a one-word tag to each memory item. Their testing showed that adding tags increased memory usage by as much as 61%, and slowed their RBMM-only system by up to 30%. They found that adding RBMM to a GC system with tag words improved execution speed by up to 42%. This improvement is the result of being able to free some of the memory without the overhead of GC.

Elsman investigated type safety in the combined RBMM and copy-collector system implemented for Standard ML [26, 38]. The combined system allows pointers between regions to exist. A dangling pointer can be created when a reference between two regions occurs. When the newer region is popped off the region stack, the older region will contain a pointer (dangling) that references newly-reclaimed memory. The author introduced pointer safety, and proved the soundness of the implementation, by eliminating dangling pointers in the region typing system.

In 2002, Elsman introduced a partially tag-free garbage collector for a Tofte and Talpin typed RBMM system implemented in Standard ML [25]. This concept is similar to our approach as it uses a “Big Bag of Pages” solution for tagging, or applying type information, to a particular allocated item. By using such a technique, type information does not have to be presented along side every allocation, rather what region the allocation belongs to can aid type-inference. Less tag can reduce the amount of memory necessary to represent type information of an allocated item. The GC that we introduce can decide the type for all allocated variables and this is known at compile time and encoded in the address based on the region from where the allocation was produced from.

In 1998, Christiansen and Velschow investigated adding region semantics into a Java-like language they designed called “RegJava” [13]. Their system was inspired by the Tofte approach for ML, thus RegJava uses stack-based region allocation. However, their system does not implement region inference
statically, but relies on programmer annotations for managing regions. In many cases, their system is able to improve memory use over that of GC. Their system also has predictable memory use, which is common for RBMM systems, since allocation and deallocation are explicit; deallocation is not dependent on a GC strategy that might take a non-predictable amount of time.

Predictable and real-time languages are necessary for mission critical and/or embedded devices. In contrast to GC, RBMM is by its very nature deterministic with respect to memory allocation and deallocation, since the compiler can insert deallocation calls based on a static analysis. Having the knowledge of when a variable escapes a function is critical for calculating how long an object can live. Salagnac, et al. [68] implemented a “fast” static analysis to aid their goal of implementing RBMM on an embedded Java framework. This static analysis can be used to implement a semi-automated region inference algorithm [69]. Their system detects memory leaks which then produces feedback to the developer, in order to prevent what the authors call the, “region explosion syndrome.” The latter is a problem of RBMM systems, whereby all objects are allocated from a single region that lives for the lifetime of program execution. This is analogous to a memory leak, and has also been noted in the Tofte approach [76]. We call this the region bloat problem. The authors found that their benchmarks produce regions of short lifetimes. This is ideal, since memory is constantly being recycled in a deterministic manner. The contrast are longer-lived regions, which can occupy a large portion of the memory space and can become candidates for GC (along with the execution cycles required to GC). Salagnac et al. pointed out that “for most programming patterns” their memory consumption was on par with that of GC. However, they did find a class of programs which performed worse and acted as if they had memory leaks, exposing a limitation of their analysis.

Another Java implementation using RBMM is that of Cherem and Rugina [10]. Like Salagnac et al., Cherem and Rugina’s approach is based on a
points-to analysis and also suffers from the memory leak problem mentioned earlier [69]. The authors found that short-lived regions benefit memory utilization, and that many allocations could be placed on the program’s stack, which reduces the need for heap allocation.

Boyapati, Salcianu, Beebee, and Rinard [9] also investigated implementing RBMM in a real-time Java framework capable of handling multi-threaded applications. Their approach utilizes combined regions and ownership types. All objects are allocated from a region, and have an owner (either another object or region). Ownership types associated to objects are used to generate an ownership hierarchy-tree during analysis, whereby an object can only be accessed by an owner. Their system, as with the Gerakios et al. approach [30], forms a hierarchy of regions and is safe from dangling pointers. This system also allows multiple processes to share region data.

Chin et al. [12] also implemented a system of region inference for Java which is based on the Real-Time Java specification (RTJS). Their approach uses a stack of regions and never creates dangling references. The authors compare their fully-automated RBMM system with that of a manually annotated region system. The authors found that the contrasting manual approach to memory management “may represent a sizeable mental effort for a programmer with only a region type checker.” For the set of applications measured, their automatically annotated system performs just as well as the manually annotated version of the applications.

The Real-Time Java specification defines a lexically scoped region system to reduce the amount of time spent garbage collecting and to improve system predictability [40]. This system relies on programmer annotations to define the scopes within a program. Scopes impose a lifetime on objects allocated within them. As with regions, a scope cannot be freed until all of its objects are no longer reachable. RTSJ scopes rely on reference counting to prevent premature scope reclamation. Dangling pointers are eliminated by preventing older objects from referencing objects with shorter lifetimes. Nested scopes increase runtime due to the system checking references of
the scopes; reference checks are expensive. Hamza and Counsell found that “most of the benchmark applications used less heap space when using regions rather than garbage collection.” Also, the authors pointed out that reference counting imposes additional time overhead. They also found a higher space overhead when allocating short-lived objects inside longer lived scopes. This is the same as the region bloat problem discussed in Chapter 2.

Stoutamire [73] researched a similar RBMM concept for the Sather programming language to improve memory locality. His model introduced the concept of zones (regions) which map objects and threads onto hardware. Zones are organized in a tree structure and can be individually garbage collected. Stoutamire’s results from a partial implementation of his zone model favors zones for speed, in most cases, over a non-zone model. The author noted that more study needs to be conducted on his model to conclusively say that a zone-based system is ideal for practical applications. To clarify, Stoutamire’s zone model is different from what our design presented in this thesis. First, our system is not designed to map directly to hardware. And second, our zone design is used to partition a region into memory per data type. Stoutamire’s zones can be any data type and contain two different memory allocators: one for objects that do not contain pointers and another allocator for objects that have pointers. In contrast, our allocator will allocate from the zone in a region for the requested data type. Our zone model is only used when we compile our system to use our region-aware garbage collector.

Gay and Aiken [28] found that their manually annotated RBMM system for C, using their C@ library, can have performance comparable to manual memory management in both space and time, while also being better (in many cases) than a conservative garbage collector. Their approach relies on reference counting to prevent premature region reclamation. This count reflects the number of other regions and variables that point into the region within question. The authors found that maintaining these counts was expensive and could comprise anywhere from “negligible to 17% of runtime.”
The authors later improved upon C@ and its reference counting overhead in their later research of RC (a region compiler to replace C@) [29]. In contrast, our protection counter is a per-region counter. Our system prevents premature region removal by only incrementing this counter when a function call is made and the caller needs a particular region at a later time.

The safe dialect of C, Cyclone, implements a semi-automatic RBMM system, requiring programmers of multi-module programs (programs consisting of multiple translation units) to manually insert some region annotations [37]. Cyclone does provide automatic default annotations. The system also contains a garbage collected region for managing reclamation of memory allocated from traditional manual allocations (e.g., malloc). The system’s region and control flow analysis is designed such that dangling pointer access is a compile-time error. All of their results had slightly longer running times than C versions. This finding is expected, since adding bound and null-pointer checks produces additional runtime overhead. This overhead was not significant in the case of non-compute-intensive (e.g., an HTTP client) applications; however, it was significant (from 2.07-2.85 times slower) for some compute intensive benchmarks.

Lattner and Adve [55, 56] presented an automatic C based implementation of RBMM for their LLVM system. They found that 25 of the 27 benchmarks they tested ran faster using RBMM, or “pooled allocation”, compared to using malloc.

The logic programming language Mercury was implemented with an automated RBMM system, in contrast to its existing garbage collector [64]. The system’s stackless design was inspired by Cherem and Rugina’s Java RBMM implementation, to permit shorter-lived regions [63]. On a set of benchmarks the system ran 25% faster using RBMM than with the Boehm GC [65].

Parallel processes and threads can complicate how data is accessed, and consequentially poses challenges for RBMM. When there is only one process, the order in which a program accesses memory is trivial and predictable.
Therefore, a compiler can analyze and safely deduce when memory accesses can occur. With parallel computations, there is not necessarily a deterministic means of knowing when a process might access the same piece of data that another process is using. Here the term “access” refers to memory reads and writes.

Gay and Aiken [28] mention that implementing RBMM in a parallel system should be trivial, as locking (to prevent concurrent data mutation) only needs to occur during the region creation and removal operations.

Seidl and Vojdani [70] use region analysis, not for memory management, but to enhance an interprocedural static analyzer, GobLint. The latter detects race cases in C programs, and has been used to detect data races in portions of the Linux kernel.

Gerakios, Papaspyrou, and Sagonas discuss the capability of using a tree-based hierarchy of regions to facilitate concurrent programming and parallelization for Cyclone. In that system each region contains locks, which are common in traditional non-region-based parallel programming [30, 31]. If a process has a lock on a memory resource, or region, no other competing process can access that data until the lock is released. The results of their hierarchical region locking show a mix in favor of both the unmodified and modified executables. The runtimes were nearly double in three of the five benchmarks measured. The authors attribute the penalty in one of their benchmarks to using multiple locks on a single data structure and its elements. Ultimately, there is room for improvement. The key point is that they implemented a safe parallel system using RBMM.

Lattner and Adve [55] mention parallelization for their C/LLVM implementation of RBMM. They note that if the compiler can determine that two data structures are disjoint and have no shared references, it is possible to parallelize writes to these structures.
In this chapter we conclude our exploration into automatic memory management by looking at some future work that can augment our Go-RBMM implementation. First we look at optimizations that can provide a more time and space efficient system than what we have already presented. Then we review what this thesis has covered, and the work we have accomplished.

8.1 Summary

This thesis has explored the capabilities of RBMM and a novel combination of RBMM with GC for the Go programming language. We have discussed the language syntax, concepts of memory management, and provided an exploration into using RBMM as a method for automatic memory management. The system we have developed requires no programmer code annotations aside from what the language already provides (new and make). Thus, we maintain the simplicity for the programmer that Go provides; the programmer is saved from spending much time reasoning (and possibly making poor decisions) about memory management. We have shown that we can extend an existing language to support RBMM via use of a GCC plugin and a modified runtime library, which can be linked with the Go program being compiled.

In Chapter 3 we introduced RBMM concepts and our design. We based our presentation on a simplified Go/GIMPLE syntax large enough to cover the interesting aspects of Go. Our goal was to use an RBMM system to
reduce time and space overhead compared to Go’s existing GC. Our evaluation suggests that using RBMM is comparable to unmodified Go benchmarks running with their garbage collector. For one case in particular, binary-tree, we significantly improve execution time by reducing time spent in GC by using regions. As expected, RBMM can increase the size of the binary, as a result of our code transformations inserting region operations into the program at compile-time. However we have seen that region maintenance at runtime produces little overhead. In fact, it seems that numerous small regions can be beneficial to program execution time. This can limit a program’s memory usage since regions with fewer items have a higher probability of being collected earlier than regions with many items.

Chapter 4 introduced a formal semantics to reason about memory management correctness. We used these semantics to prove the equivalence of our RBMM system to that of the original Go system.

Although we are not the first to combine RBMM and GC, as far as we know we are the first to combine a region-specific garbage collector that operates directly on regions [18]. Our hope was to eliminate the region-bloat problem by removing unreachable items from within regions by designing a region-aware copying garbage collector. We presented such a design and partial implementation in Chapter 5. With further improvement, our collector can be extended to collect subsets of a program’s memory space, that is, act as an incremental collector operating on a subset of the program’s regions. This modification should reduce the runtime overhead of our region-aware garbage collector. We also provided a design for supporting reclamation of sequences of items within Go slices.

We concluded our exploration by introducing and implementing a design to support RBMM within the CSP parallel context of Go. We were able to show comparable time and space measurements on the short-lived benchmarks we evaluated.

Memory management provides a large research domain filled with exciting areas to explore. The ideas we have presented in this thesis can be
applied not only to Go, but to other languages as well. We hope that future researchers can benefit from our research and will continue explore this topic area in other unique ways.

8.2 Future Work

We have discussed an implementation of RBMM for the Go programming language. We now look at a few optimizations that can improve the system we have designed, but were left out due to both implementation complexity and time constraints.

8.2.1 Optimizing our Garbage Collector

Garbage collectors are notoriously difficult to program, and we can attest to the fact that they are complicated and time-consuming to debug. In Section 2.3.7 we discuss the complications of creating a memory management system. Additionally complicating matters is the fact that our region-aware GC system is a combination of two automatic memory management techniques, RBMM and GC. We have left out many optimizations in our system in hopes of producing a more stable environment, which we can later optimize.

Zone Header Optimization

One optimization that we expect to be beneficial is to improve how our runtime system handles the creation of zone headers. Currently, when our runtime system creates a new region, it will allocate enough room to store a zone header in that region for every type in the program. This is overly conservative, but made our analysis quite simple. Every time our analysis discovers a memory allocation, we can simply specify an index into the region’s zone header table to allocate from and have no need to resolve where the zone header resides at runtime. Since all regions have all zone
types, the index will always be the same irrespective of the region used. Our analysis cannot always determine which region will be allocated from at compile-time. Letting all regions have headers for all types simplifies things considerably. Unfortunately, this also means that many regions will have unused zone headers, which can take up quite a bit of memory. A more optimal solution would be to have a mapping from type information to zone header. This table would have to be unique per region and used at runtime. The lookup should not be computationally expensive, and can be keyed on the type information, which we know at compile-time for every allocation.

Another possible solution to the aforementioned zone header resolution problem is to make the region’s zone header table hold pointers to zone headers that are lazily instantiated. This pointer table would have a pointer for each possible zone type that can exist in the program (similar to what we do now but using pointers instead of populated objects). Instead of actually allocating memory for the entire zone header, this optimization should only allocate the header’s memory when the program makes an allocation request for a type and the pointer in the region’s zone header table is NULL. This approach will still suffer from having unused data (the unused zone pointers), but the cost will be reduced to one word per unused zone type.

**Incremental Collection in Regions**

The algorithms discussed in Chapter 5 describe a garbage collector that is capable of collecting from a subset of all regions. Such an incremental design can reduce collection times, resulting in a faster running program. Incremental collection can work because an object’s address can be used to determine what region it belongs to. Therefore, a region is capable of knowing what other regions its objects point into. With this information, a garbage collector can be made capable of only scanning the subset of all regions in the process. For simplicity, our current implementation garbage collects all regions. We believe that our collector can be optimized to only scan regions that hold pointers (directly or indirectly) to data in regions.
being collected.

Optimal GC performance requires a highly tuned collector, and we have not spent much research time exploring when to run our collector. We have not considered it a major focus of our work and it would take a considerable amount of testing and research to find the optimal choice of when and what to collect.

**Slice Sequence Garbage Collection**

When our collector visits a slice item, it preserves the entire slice and does not attempt to accomplish the more complicated task of preserving series of items from the slice, as described in Section 5.5. Handling slices has been one of the more challenging aspects of writing a region-aware garbage collector, and we were not confident optimizing things any further without having a stable baseline to work against. The obvious optimization here would be to allow slices to have sub-sequences of items collected. This may or may not prove to be much of a benefit and would be program specific.

### 8.2.2 Applying our Research to Other Languages

Our implementation exists as a plugin for GCC with an accompanying object file (runtime system). Our plugin’s analysis and transformation passes operate on GCC’s middle-end intermediate representation languages (GIMPLE and RTL). Since all front-end language parsers to GCC will produce a parse tree that GCC converts to GIMPLE, our system can be modified to handle other languages that GCC supports. Of course, the runtime system will also need to be modified to work specifically with the source language’s runtime system.
Appendix A

Go Programming Language

Simplicity is a great virtue but it requires hard work to achieve it and education to appreciate it. And to make matters worse: complexity sells better.

Edsger W. Dijkstra

This chapter introduces the Go programming language which is used throughout this thesis as a vehicle for exploring the concepts of memory management. The language is not covered in its entirety; however, enough background is provided in this chapter so that the examples and language-specific topics discussed later make sense. This thesis uses code that is compatible with version 1 of the language. We assume that the reader is familiar with the syntax presented in languages such as C/C++.

Our RBMM proof-of-concept does not transform Go code directly, instead it transforms GCC’s intermediate representation, GIMPLE, of a Go program. This abstracts away many of the complexities of the higher-level language and presents our analysis with a three-address-code representation in static single assignment form that our system can analyze and transform.

Section 3.2 introduces a distilled syntax that we will use throughout the remainder of this thesis. This syntax covers features of Go that are interesting from the point of memory management. The reader who is already familiar with Go should skip to Chapter 2.
A.1 Introduction

Go is an imperatively-styled, statically typed, general purpose system level programming language that provides concurrency primitives, interfaces, and higher-order functions. The language was created by Google around 2007, and released to the public in November of 2009 under an open source license [74]. It is a procedural language, similar to C, which facilitates quick development and eases analysis [36]. Its strong and statically typed syntax does not require cumbersome code annotations, and allows programmers to create their own data types and interfaces.

The Go runtime system utilizes a garbage collector to aid memory safety. The collector relieves programmers of the burdensome task of managing the reclamation of memory, which would otherwise compromise program safety due to improperly used resources by programmers, such as memory leaks.

Programs written in Go can utilize simple and safe parallelization concepts to enhance application scalability. Functions are first-class objects, which can be used for exception handling, or for post-function cleanup (e.g., closing file handles if a function returns early).

The Go language comes with a suite of libraries providing programmers with a variety of utilities: such as cryptographic, networking, and http/web related functions.

A.2 Go Syntax

The syntax of Go is very similar to that of traditional imperative languages like C and C++. The syntax is non-ambiguous and non-obtrusive, while still maintaining enough information for the compiler to perform a strong typecheck of the source code.
A.2.1 Declarations and Assignments

The variable and function declaration syntax is similar to C’s, but with the type and identifier names reversed. This order can ease analysis for compilers and other parsing utilities, as well as remove declaration ambiguity. For instance, the following block of code illustrates a basic function in Go.

```go
func AddTheUniverse(x int) int {
    universe := 42
    return x + universe
}
```

This example declares a function `AddTheUniverse` that has one formal parameter, `x`, and returns an integer value. If the declaration is read aloud left to right, the meaning becomes clear to the reader: “Function AddTheUniverse takes `x` as an integer for input and returns an integer.” In the body of this function we declare a variable `universe` which has the value of 42. This is a mutable variable. We could have also declared `universe` as `var universe int`. Since the compiler can infer the type of the variable `universe` we can use the shorter syntax which combines variable declaration and assignment `:=`. The latter is useful for declaring temporary variables that only require scope within a block of a function, such as iterators in loop constructs:

```go
for i := 0; i < 100; i++ {
    // Do stuff
}
```

In the latter example, `i` only has scope within the loop, use of `i` outside the loop will result in a compile-time error.

To further illustrate the simplicity of the declaration syntax, consider a more complicated case:

```go
var x []*Thing
```
Again, if this line is read left-to-right the meaning becomes clear: “variable x is an array containing pointer-to Thing objects.”

### A.3 Modules and Imports

Go comes with a large suite of libraries, also called modules. There are no source include files, instead the programmer expresses that they want a module included in their program via the import syntax. This speeds up compilation since there is no need to transitively parse a header file and all of the header file’s included headers.

The compiler will only permit functions, types, and global variables declared public, as being accessible via import. To declare something private, the global variable identifier, type, or function name must begin with a capitalized first character.

Modules are used in the program by prefixing the global variable, type, or function that the module provides with the module name.

```go
package main
import "os"
func main() {
    file, err := os.Open("myfile.txt")
}
```

This example imports the os module, and call the public routine it provides, Open. Modules are created by using the package statement as the preamble in the source file. In the case of the “os” module, all of the source files that make up this module begin with the statement: package os. The main package is used for creating an executable and not a library/module.
A.4 Types

A.4.1 Common Primitives

Go has a set of primitive data types to represent boolean, integer, floating point, and complex number information. Except for bools, these types can be suffixed with their bit size (either 8, 16, 32, 64). For instance, int8 represents a 1-byte integer, uint16 represents a 16-bit unsigned integer, float64 represents a double (64-bit) wide floating point variable, and complex128 represents a complex number with real and imaginary parts consisting of 64 bits each. There is no char data type to represent a 1-byte generic value, rather that is what the byte data type is for. While the language developers avoided ambiguity between float and double sizes, by not defining a double type and forcing the programmer to use a more meaningful type-name consisting of bit-size (float32 or float64), they did not avoid this ambiguity for integers. While integer types can be defined with a suffix, as described above, they can also exist without the size suffix, such as: int and uint. These types are either 32 or 64-bits in size, based on the machine’s architecture. The specification does not explicitly mention that their lengths are based on architecture, rather that an int is the same size as a uint and that the latter is either 32 or 64 bits in size [34]. The uintptr represents an unsigned integer long enough to store a pointer (address).

A.4.2 Pointers

The Go language includes pointer variables but does not support pointer arithmetic. Pointers can alias the same piece of data; however, a pointer’s value cannot be manipulated based on algebraic operators. This feature provides type safety, which prevents the pointer from accessing data that might be of another type. Manipulating data that is not of the type that the pointer was declared as can crash programs or produce incorrect results.
The above example declares \( x \) as a variable holding the integer value of 42. It then declares \( y \) as being a pointer to \( x \). The \( * \) operator is used to dereference \( y \) and obtain the value that it points to, 42. In C the programmer can manipulate what \( y \) points to via pointer arithmetic such as: \( y = y + 1; \). This is not permitted in Go, as it would compromise type safety.

### A.5 User Defined Types

The \texttt{type} keyword is used for either creating structures, interfaces, or aliasing a type. Go does not provide classes or inheritance, rather the programmer can simply create structures and utilize interfaces. If any structure type has all of the methods defined by an interface, then that structure type is said to support that interface.

To begin with, the \texttt{type} keyword can be used to redefine a type.

\begin{verbatim}
    type point int [2]
\end{verbatim}

The example above introduces a type synonym, \texttt{point}, which is an array of two integers.

#### A.5.1 Structure Types

Structures consist of just field declarations. Methods for the type are declared and defined as their own functions, and are not specifically mentioned in the body of the \texttt{struct} definition. A leading capital-letter in used defined types, methods, field names, and global variables denote public access to the data. This is used to support information hiding in a module.

There is only one selector, ".", used for accessing fields. Unlike C, the indirection operator, "->", is not used to reference a value from a pointer,
instead "." is always used.

```go
var x Thing
var y *Thing
y = &x
total := x.someField + y.someField
```

In this case the `someField` field of `x` and `y` is obtained. The compiler knows if the variables are pointers or not and will automatically insert the dereference operator required by `y`’s type. This relieves the programmer from having to remember such information.

The following example defines a public `Cat` type that has an age field, which is private.

```go
type Cat struct {
    age int  // private age field
}
```

We extend the capability of our `Cat` type by defining a method for it named `MatingCall`:

```go
func (c *Cat) MatingCall() {
    println("'Purr'")
}

func main() {
    garfield := new(Cat)
    var morris Cat

    garfield.MatingCall()
    morris.MatingCall()
}
```

This `MatingCall` method can be called on any pointer and non-pointer instances of the `Cat` type. When the method is invoked, it will have a pointer to the `Cat` instance called `c`. The compiler is smart enough to know that
even though *morris* is not a pointer, its address will still be passed to the `MatingCall` method when the *morris* instance invokes it. Similarly, if the method were to be declared as `func (c Cat) MatingCall()` with `c` not being of pointer type, the compiler will produce code having the same effect as a call-by-value function call.

### A.5.2 Interfaces

Interfaces provide a duck-typed syntax for allowing multiple types to be treated as a more generic type. An *interface* declaration just specifies the function prototypes that a member of the interface must fulfill. If a type has all of the methods defined by an interface, then that type is a member of the interface. In other words, if it looks like a duck and walks like a duck, it is probably a duck.

```go
type Cat struct {
    age int   // private age field
}

type Duck struct {
    name string // private name field
}

type Animal interface {
    MatingCall()
}

func (d *Duck) func MatingCall() { println(‘Quack!’) }
func (c *Cat) func MatingCall() { println(‘Purrr!’) }

func Wild(a *Animal) {
    a.MatingCall()
}
```
func main() {
    var thing *Animal

    thing := new(Cat)
    Wild(thing)

    thing = new(Duck)
    Wild(thing)
}

Both Cat and Duck types are members of the Animal interface. Therefore, functions and pointers can generically refer to an Animal instance instead of the specific type. The Wild function operates on any object that satisfies the Animal interface. This function calls the MatingCall interface-defined method on those objects. The result of running this example would be “Purrr!” and then “Quack!”.

The empty interface, interface{}, is the most generic means of working with types in Go. All primitive and user defined types fulfill the empty interface type, and this is how the println built-in routine is defined. It can accept any type it is passed; however, the compiler will reject types it does not know how to print. The term built-in refers to a language-provided programming feature, such as a datatype or routine.

A.6 Container Types

Go provides a series of built-in container types for use by the programmer. This section discusses these types.

A.6.1 Arrays

An array represents a series of objects or primitives in Go. Interfaces, including the empty interface, can make up the elements of an array. Arrays are declared statically and their length must be a constant. Go performs
bounds checking on arrays at either compile and run times. Since Go is call-
by-value, arrays can impart quite a memory overhead, since the passing of
an array to a function will require that the program copy the entire contents
of the array to the formal parameter in the callee.

```go
func processArray(vals [55]int) {
    // Use the vals array
}

func main() {
    var myarray [55]int
    processArray(myarray)
}
```

In this example 55 integer values are copied from the caller, `main`, to
the callee, `processArray`. Any mutations to the elements of the array by
the callee will never be seen by the caller. For somebody familiar with C,
where arrays are passed by address, this may come as a surprise. Passing
an entire array by value is both computationally expensive and memory
expensive, since the CPU must duplicate the contents of the array. This can
be avoided by passing a pointer to the array, or by using slices, which are
passed by reference.

### A.6.2 Slices

Unlike arrays, slices are instantiated dynamically either through the `make`
keyword or by converting an array to a slice. A slice can be thought of as a
vector: an array which can shrink or grow at runtime.
This example declares a slice containing 55 integer elements. Since slices are passed by reference, their associated overhead is low compared to the copy-by-value that an array would cause. Portions of slices are still considered a slice and are specified via the "::" operator: where the value to the left of the operator gives the first element in the slice, and the value to its right gives the index of the first element after the slice. In other words, the lower bound is inclusive, but the upper bound is exclusive: \texttt{s[m:n]} represents a sub-sequence of the values in array or slice \texttt{s} starting at index \texttt{m} and ending at index \texttt{n-1}.

In this example, all assignments will create slices. Just passing empty operands to the "::" operator will create a slice with all of the array’s contents. Not specifying either the upper bound or lower bound will default the operand to be either the array’s first element or the \texttt{n-1} element respectively. Updates to the slice contents will update the array that the slice derived from.
A.6.3 Maps

Maps are another built-in data container type provided by Go; they map keys to values. The keys and values can be of any type: primitive, user defined type or interface. Like slices, maps are automatically passed by reference, therefore there is no need to prefix a map-type argument with the address-of “&” operator.

```
var mymap map[int]string
greetings := map[int]string{1:"Hello", 2:"Yo"}
colors := make(map[float32]string, 100)
```

The first line in the example above declares a map that maps integers to strings. The map is declared as having a key of type `int` and values of type `string`. The second line declares and defines a map, filling it in with two key-value pairs. The third line shows the creation of a map using the `make` allocator to create an initial map that can hold 100 keys and their values. Except for the example in the first line, the runtime system will automatically extend the map if a value is added to a key that was not previously stored in the map. This rule results from the fact that the compiler will actually create both the maps `greetings` and `colors` dynamically, even though the programmer never created `greetings` via `make`. Since `greetings` is initialized, it can be extended. On the other hand, `mymap` is never dynamically allocated. The compiler will actually create `mymap` as a pointer to `nil`. Therefore, it never has any initial data, and cannot be extended. If `mymap` is set to alias `greetings` via assignment, `mymap = greetings`, then it can be extended following that assignment.

A.7 Memory Management

This section covers the dynamic memory allocation routines provided by the Go language.
A.7.1 New and Make Allocators

Go provides two keywords for allocating dynamic memory (similar to `malloc` in C), namely `new` and `make`. To create a pointer to allocated memory representing a particular data type, the `new` keyword is used.

```go
mypointer := new(Thing)
```

The above declaration creates a pointer to an allocated item of type `Thing`. Go will zero-initialize all allocations, so there is no need to clear the returned allocated data.

The `make` keyword is an allocation built-in function that is used for dynamically allocating instances of slices, maps, or channels. We will discuss channels in Section A.11. The `make` routine has three parameters. The first parameter specifies the data type to create, the second specifies a `length`, or number of elements, that values of the type can contain, and the third specifies a `capacity` reserving additional room for the elements. The `length` parameter is required for slices, and `capacity` is optional in all cases. `capacity` and does not limit the bound of the data type. However, specifying a `capacity` can reduce the overhead of expanding a slice or map at a later time. Both the `length` and `capacity` parameters are optional for channel types. Values for container types that are allocated with `make` can grow past their initial capacity to hold more data.

Garbage Collection

The Go 1.0 release provides a stop-the-world parallel mark-sweep garbage collected runtime environment. Programmers do not need to worry about deallocating memory after calling `new` or `make` since the language implementation will automatically release memory that is no longer used by the program. It should be mentioned that the Go development team is working on a more efficient collector, as the one provided by the 1.0 release is relatively basic.
A.8 Control Flow

Go provides a variety of mechanisms to alter the control flow of a program. Here we list the standard mechanisms. Section A.10 discusses a Go-specific escape mechanism.

A.8.1 If-Then-Else

The if-then-else constructs in Go are similar to C's.

```go
if x == 42 {
    println("This sentence is not exciting!")
} else if x == 43 {
    println("This sentence is false")
} else {
    println("This is a sentence")
}
```

A.8.2 Switch

Switch statements in Go provide a way to branch across multiple conditions. Unlike C, the case statements in the switch can be expressions. The body of the first case-expression that evaluates to true will be executed. If multiple case-expressions satisfy the switch statement then the first one will be executed.

```go
switch x := getUniverseValue(); {
    case x > 42:
        println("The universe is expanding")
    case x < 42
        println("The universe is contracting")
    case x == 42:
        println("Just right!")
    fallthrough
    default:
```
println("The universe is a hologram.")

Also unlike C, cases in the switch do not require a break: by default, cases do not fallthrough. Instead, if a fallthrough is desired, the programmer must use the fallthrough keyword.

## A.8.3 Loops

The only looping construct, aside from using goto or recursion, is the for loop. The range keyword produces two values, an index number, and value. This provides a convenient way of iterating across values of the built-in container types (slices and maps).

```go
for i, v := range myslice {
    // Do stuff
}
```

The `i` variable is the iteration number, which can be used as an index into the slice, and the `v` variable is a copy of the value contained in the array at the index `i`.

For loops in Go can behave like while loops in C.

```go
for i < 1000 {
    // Perform magic
}
```

As with C, a for loop in Go can also emulate for loops in C, with initialization, termination condition, and reinitialization:

```go
for i:=0; i<1000; i++ {
    // Perform magic
}
```

A for statement without a terminating condition can act as an infinite loop. As in C, a break statement escapes the encapsulating loop, and a
continue statement returns execution to the beginning of the encapsulated loop.

## A.9 Higher-Order Functions

Functions in Go are first class objects. They can be passed as arguments to functions, stored in variables (including maps, slices, and arrays) and also returned from functions.

```go
func calculate(f func(a, b int) int, vals []int) int {
    total := 0
    for i := 0; i < len(vals); i++ {
        total = f(total, vals[i])
    }
    return total
}

func main() {
    add := func(a, b int) int { return a + b }
    vals := []int{200, 300, 400}
    calculate(add, vals)
}
```

In the previous example, the `calculate` function takes another function as input. That input function takes two `int` values \((a, b)\) and returns an `int`. The function `main` creates a function `add` which sums two values, and constructs a slice containing three values. It then calls the `calculate` routine to compute a running sum over the values in the slice using the function it was passed, `add`.

Anonymous functions with state (closures) can be created as well. Examples of such appear in the next section on the `defer` statement.
A.10  Defer

A *defer* statement is a closure that is executed just before the function it is defined within returns, but after the return expression has been evaluated. Deferred statements are always executed, even if the function returns via an explicit return statement, and not by reaching the end of the function. These statements are similar to C++ destructors or Java finalizers, and permit the program to reclaim resources when an object goes out of scope. However, in Go, *defer* operates at the function level and not object level. This feature can be handy in cases such as file input/output where a function opens a handle, but returns early without closing the handle. Deferred statements can also modify the function’s return value.

```go
func loadFile(name string) {
    file, err := os.Open(name)
    if err != nil {
        os.Exit(-1)
    }

    defer file.Close()

    if maybeFail() {
        return
    }

    file.Close()
}
```

In this example, `file.Close()` is registered as a deferred function, which will be executed upon function return. If the `maybeFail` routine returns *true* the file handle will be closed since `file.Close()` was registered as a deferred function. If `maybeFail` returns *false*, and the function completes, then the `Close` will occur twice; however, that is not a problem. If `loadFile` calls `os.Exit()` then any deferred functions will not be executed.
Also of importance is the order of evaluation for arguments and variables used by the deferred statement. These values are evaluated at the time the deferred function is registered.

```go
func foo() {
    x := 1
    defer println(x)
    x = 2
}
```

Even though the body of the `defer` is executed after the last statement in the function body, where `x` contains the value 2, the deferred `println` will print a 1 to the output. This occurs since `x` was evaluated as 1 when the deferred function was registered.

In effect, the `defer` statement creates a closure containing a snapshot of the then-current values of the relevant variables, and is permitted to modify the enclosing function’s return value.

A function can have multiple `defer` statements. These are stored in a stack, so the most recently registered will be executed first. The following example illustrates this stack discipline, while also showing that deferred statements can be defined as anonymous functions.

```go
func foo() {
    defer func() { println("Three") }()
    defer func() { println("Two") }()
    defer func() { println("One") }()
}
```

This example would print the strings “One”, “Two”, “Three”.

### A.11 Concurrency

Parallel computation is an inherently tricky task. To squeeze the most performance out of a modern machine with multiple threads of execution and
CPU cores, programmers must rely on tricky techniques so that data can be shared between all threads of a program. This is a notoriously complicated feat for most humans. Most languages are designed with a single thread of execution in mind, that is, no concurrency. However, operating systems and additional libraries can be utilized by the human to parallelize their computations and speed up execution. This typically requires synchronizing access to data that are shared across multiple threads of execution. To prevent non-deterministic access, such as reading data that are in the state of being mutated by another thread, humans must place locks. As with memory management, humans often make bad judgments, and introduce bugs which compromise the integrity of their programs. The Go language uses a simple and safe method of computing data concurrently based on C.A.R Hoare’s idea of Communicating Sequential Processes (co-routining) [49]. Similar to Erlang and Occam, concurrent data exchange and communication between threads is accomplished via named channels. This method of synchronization reduces the need of the programmer to make explicit synchronization calls.

Go processes are referred to as go-routines. Go-routines are functions that execute concurrently with other go-routines (including the main thread of execution). They are light-weight threads: cheap to create and cheap in terms of memory usage. A single operating system thread can execute multiple go-routines concurrently. If any thread is blocked, the go-routines on another system thread will still continue their execution [35].

The go keyword can prefix any function call, causing it to run as a light-weight thread in parallel with other threads, including the main thread of execution.
In the example above, the call to `greeting` will be executed in a separate go-routine from `main`. Since `main` will probably terminate before `greeting` has time to call `println`, the output will never be seen. This is not guaranteed, but chances are that `main` will terminate before the call to `println`. If a delay of sufficient time was to be introduced just after the go routine execution, then the probability that `greeting` will complete is increased. However, this is non-deterministic. To prevent further execution, until the data/function has been processed, a channel can be used. Channels are used to transfer data between concurrent computations, such as between `main` and `greeting` in this example. Synchronization can be facilitated through the use of channels.

```go
defun greeting (ch chan int) {
    println("Hello!")
    ch <- 1
    println("World!")
}

func main () {
    ch := make(chan int)
    go greeting(ch)
    <- ch
}
```

This code expands on the previous example by introducing a channel variable. This will establish a communication between the `main` thread and that which is executing `greeting`. The "<-" operator is used to send or receive data from a channel. In the modified example, a channel is first created in
main via the make built-in function. That channel is passed to greeting, and main waits until any data is sent back down the channel. Notice that greeting has no return value. The sending of data down a channel does not mean that the go-routine has completed, but it does mean that the routine is in a state whereby it has data ready for any other go-routine listening on the channel. In this example main will continue executing as soon as it gets data (which it ignores). In this example, what greeting sends down the channel is arbitrary. The data could have just as well been any other int value, as that was the type we associated with the channel when it was declared. Since main blocks until something is sent down the channel, then immediately terminates, it is unlikely that the “World!” greeting will be displayed. In other words, “Hello!” will be seen, but as soon as main continues in its separate thread, it will terminate, possibly before “World!” is displayed.

As mentioned, make is used to construct a channel. If no count parameter is passed to make then the channel is considered unbuffered. However, a buffer can be specified by giving a count for number of items that can be stored in the buffer:

```
ch := make(chan bool, 100)
```

The line above would allocate a channel that can store 100 boolean values. For instance, if the channel is buffered and 100 items have been stored down it and no receiving go-routine has read any of the data from the channel, then the sender will block until there is room in the buffer. Buffers are first-in-first-out (FIFO) queues. This means that the receiver of the data will be able to process all information it is passed from the sending go-routine, irrespective of how slow the receiver might be.

## A.12 Compilers

There are two primary compilers following the Go 1 specification of the language, the Google compiler and GCC (gccgo). The Google compiler
comes with a suite of utilities all accessible via the *go* tool. This utility not only builds go programs, but eliminates the need for Makefiles. The command assumes a predefined directory layout where it can locate and build source code and assemble modules if a project consists of multiple modules. This utility also formats source code, can download build and install external libraries, and can run benchmarking and tests. The *go* tool can be selected to use the GCC compiler if desired.

### A.12.1 Google

The Google compiler was originally based off the Plan9 C compiler. This compiler was designed to build programs fast while also making binaries portable via static linking. The downside of fast compilation is that the compiler performs fewer optimizations, resulting in an executable that does not always run as fast as what other optimizing compilers (*e.g.*, *gccgo*) might provide.

### A.12.2 GCC

While not as fast in compile-time as the Google compiler, *gccgo* can produce smaller and highly optimized binaries. The *gccgo* front-end for GCC inputs Go source code and translates it into GCC’s intermediate language, GIMPLE. This three-address-code generic representation is then used in a series of optimization passes. The result is converted to the desired machine architecture and output for assembly. We use GCC to test our RBMM implementation described later in this thesis. Since GCC supports a plugin feature, it is relatively simple to analyze and transform the GIMPLE representation of a program, without the need to perform a recompilation of the compiler. Since we are analyzing and transforming GIMPLE intermediate language (and in some parts the machine intermediate language RTL), our concepts can be extended to other languages that GCC can input (*e.g.*, C, C++, Fortran, etc.).
Both the Google and gccgo compilers use the same runtime system provided by the Go language.

This thesis makes use of the GCC compiler, and its plugin feature, to explore the internals of the Go language, analyze source code, and to transform the input program.
Bibliography


