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

August 2013

Department of Computer Science
The University of Melbourne

Abstract

Memory management is a complicated task. Often the language itself exposes 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 prematurely terminate in error. 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 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 the less common region-based technique performs a static analysis during compilation and determines program points where the compiler can insert memory reclaim operations. Both automated options have their 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. Region memory often requires objects to remain resident in memory longer than garbage collection, even though such objects might not be needed.

This thesis investigates the alternative and less common form of automated memory management within the context of the relatively new imperatively styled language Go. We investigate both techniques and a way of combining the two in hopes of achieving the benefits of a combined system without the drawbacks that each automated technique provides alone. We finally investigate the sequential process communication aspect of Go (CSP) and how we can introduce a region-based memory management system that operates within a concurrent context.

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 OR the thesis is [number of words] as approved by the Research Higher Degrees Committee.

Matthew Ryan Davis

Preface

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 advisers. Our collaborative research, weekly meetings, and impromptu brain storming sessions are what follows.

Chapter 4 and Chapter 5 are based on draft papers that I had originally written discussing the implementation of a region-based memory management system for the Go programming language. While all of the source code is mine, or extracted from GCC, the ideas are the result of research and collaborations between myself and advisers. These two chapters derive from our publications to the ACM SIGPLAN Workshop on Memory Systems Performance and Correctness 2012 and 2013 respectively. Chapter 6 was extracted from our first publication and extended in this thesis, this was never in my original draft. My original drafts went through numerous rewrites by my advisers, and were expanded to add ideas that never made the initial drafts, such as handling goroutines which has now become the body of Chapter 6. This research process has been an incredibly rewarding experience, and I have learned quite a lot through this collaborative effort.

Contents

List of Figures

Chapter 1

Introduction

Science is the belief in the ignorance of experts.

Richard Feynman

Developing software is a challenging task. Sloppy coding, the stress of fast development (deadlines), language choice, and dealing with legacy code are all complicating 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 orthogonal to the problem at hand. One such 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.

If memory is improperly managed, the software can crash or produce 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 operator's commands [64]. This was the result of a memory limitation. Simply, the system responsible for manipulating the rover ran out of its main memory (128M of

RAM), leaving Spirit not responding to NASA's commands. The solution was to simply have technicians remove unneeded files from the rover's system. If improperly managed memory can affect NASA, surely it can have an impact within other mission critical contexts.

How a programmer manages memory depends on the programming language used. Many languages (*e.g.* C, 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 crash or exhaust the system's memory resources (*e.g.* memory leaks).

One way memory complicates a programmer's job is that it can be difficult to establish the lifetime of an allocation. 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 coding. 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 to reason about, which can result in error-prone programs. The programmer must be fully aware of all functions his program calls (directly or indirectly), and if they require any memory associated with returned values to be reclaimed. This means that any functions not written by the programmer must explicitly state, in documentation, the need for memory reclaim. Global reasoning is also 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 have a global context. This idea contrasts with an easier form of memory management, automatic 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 never escapes the function it is allocated within (it is local), and is almost always managed implicitly by the compiler when the program is being compiled. Stack memory is linear in lifetime, meaning that it is often allocated at the start of the function (implicitly by the compiler) and the memory is reclaimed at the end of the function. In contrast, dynamic memory is not always localized, programmers must determine when certain 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 have been reused for another value and have completely different value semantics.

Automatic memory management aims to reduce the programmer's burden and increase software reliability. For instance, the Java programming language does not require the programmer to release memory, instead the language and its accompanying runtime environment use garbage collection (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 doing such would remove an object that might be used later in 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. GC is not always an ideal solution, as such a runtime analysis can increase execution times.

Due to the additional cost of running a garbage collector, some people might 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, often at the cost of reducing program safety (*e.g.* they

present the programmer with additional resource management tasks). These languages are commonly used for driver writing, operating system kernel development, and other low-level computation tasks. Arguably, these are also the same applications that need the highest level of safety, including memory safety.

Surely we can do better than this. 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 both being memory safe and efficient at the system level. As with Java, Go provides a GC 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, region-based memory management (RBMM). Instead of waiting until runtime to locate unreachable allocations for reclaim, an RBMM system works its magic by performing a static analysis of the program's source code and inserts memory management operations during compilation. This avoids the overhead generated by runtime memory scans that garbage collectors require.

This thesis investigates the Go programming language and answers the following questions:

- How much GC can we eliminate by performing a static analysis over the program's source code and using an alternative form of automatic memory management, RBMM?

- Can we extend an existing language, Go, and add RBMM without the programmer having any additional resource management tasks to consider?

- Can we create a automatic memory management system that avoids the negatives of RBMM and GC while attaining the benefits of both

systems?

- Can we introduce an RBMM system for a concurrent language?

This thesis contributes an investigation of adding a fully-automatic RBMM system to the existing Go language. We introduce ways of handling parallelization and compile-time automatic memory management. We also introduce a combination of RBMM and GC whereby we do not have to perform a complete scan of all of the variables in the program.

In Chapter 2 the Go programming language is introduced. This chapter is not to be an all inclusive diatribe about every aspect of the language's features, rather it provides the reader with enough background to understand the concepts and examples presented later in this thesis. Chapter 3 provides a background on memory management including GC and RBMM. Chapter 4 presents our initial investigation into adding RBMM to a subset of the Go language. Chapter 5 looks at combining RBMM and GC. Chapter 6 investigates enabling RBMM within a concurrent context. Chapter 7 is a literature review of related work. Chapter 8 concludes this thesis.

Chapter 2

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 unique domain 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. The reader is assumed to be loosely familiar with the syntax presented in languages such as C/C++.

2.1 Introduction

Go is a general purpose system level programming language created by Google in 2007 and released to the public in November of 2009 under an open source license [57]. The language benefits from an unambiguous procedural syntax, similar to C, which facilitates quick development and eases analysis [24]. Go is also known for its speedy compilation times which is a property of its syntactic structure and how imported libraries (also called modules) are treated. The language provides a strong and statically typed syntax that does not require cumbersome code annotations. Duck typing is

another feature of the language which allows programmers to create their own data types and interfaces.

Go 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 simplistic 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 1.0 language comes with a suite of libraries providing programmers with a variety of utilities: such as cryptographic, networking, and http/web related functions.

2.2 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.

2.2.1 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.

In this example we declare 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

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

body of this function we declare a variable *universe* which has the value of *42*. This is a mutable variable. Similarly, we could have 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 very useful for declaring temporary variables that only need scope within a block of a function, such as iterators in loop constructs:

```
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:

```
var x []*Thing
```

Again, if this line is read left-to-right the meaning becomes clear: "variable x is an array containing pointer-toThing objects."

Additional syntax will be covered in the following sections.

2.3 Modules and Imports

Go comes with a large suite of additional 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

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

compilation time 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.

In this example we import the module *os*, 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.

2.4 Types

2.4.1 Common Primitives

Go has a set of primitive data types to represent boolean, integer, floating point, and complex number information. Except for *bool*s, 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 [22]. The *uintptr* represents an unsigned integer long enough to store a pointer (address).

2.4.2 Pointers

The Go language also permits pointer variables, with one caveat: no 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, or dangling, both of which could result in invalid memory access.

```
x := 42
y := &x
println(``The value of what x points to is:'', *y)
```

In the above example, x is declared as a variable holding the integer value of *42*. y is then declared 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 void type safety.

2.5 User Defined Types

The *type* keyword is used for either creating structures, interfaces, or redefining 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 facilitate that interface.

To begin with, the *type* keyword can be used to redefine a type.

```
type point int[2]
```

The example above defines a type called *point* which is identical to an integer array of two elements.

2.5.1 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 *struct* definition. A leading capital-letter in the type name and fields denote public access for the type/fields. 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.

```
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. This relieves the programmer from having to remember such information.

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

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

We extend the capability of our Cat type by defining a method that Cat has, *MatingCall*, below. This *MatingCall* method can be called on any

```
func (c *Cat) MatingCall() {
    println(``Purrrrr'')
}

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

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

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* no longer of pointer type, the compiler will produce code having the same effect as a call-by-value function call.

2.5.2 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.

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

type Duck struct {
    string name // 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 in-

stead 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 them. 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.

2.6 Container Types

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

2.6.1 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, thus the length must be a constant. Go performs bounds checking on arrays at both 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.

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 perform a copy operation(s) to duplicate the data used by the

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

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

array. A solution to this issue is to either use slices, which are automatically passed by reference, or to pass a pointer to the array.

2.6.2 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 (via the *append()* built-in function) at runtime.

```
func processSlice(vals []int) {
    // Use the vals slice
}

func main() {
    myslice := make([]int, 55)
    processSlice(myslice)
}
```

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 impart. Portions of slices are still considered a slice and are specified via the ":" operator: where the value to the immediate left of the operator represents the starting index, and the value

immediately to the right represents the ending index exclusive. A slice's range upper bound is exclusive, thus the value at the last index is not included. For instance, a slice *s[m:n]* represents a sub-sequence of the values in array or slice *s* starting at index *m* and ending just before index *n*. Since ranges are exclusive, the values would span the interval *[m, n)*, or contain the values from *m* to *n-1*.

```
var myarray [55]int
myslice      := make([]int, 55)
firstFive    := myslice[:5]
skipFirstSix := myslice[6:]
middle       := myslice[10:20]
arrayToSlice := myarray[:]
```

The last line in the previous example shows how an array can be converted to a slice by just passing empty operands for the ":" operator. If the operands were specified, the range of values would have been converted into a slice, representing a sub-sequence of the array.

2.6.3 Maps

Maps are another built-in data container type provided by Go. The keys and values can be any primitive, user defined type or interface. This type provides a one-to-one mapping from the key to the data/value. 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 a value of

type *string*. The second line declares and defines a map with values. 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 will automatically extend the map if a value is added to a key that was not previously stored in the map. This exception results from the fact that the GCC 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.

2.7 Memory Management

This section covers the dynamic memory allocation built-ins provided by the Go language.

2.7.1 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 memory representing a particular data type, the *new* keyword is used.

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

The above declaration creates a pointer to an 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. *make* provides

three parameters. The first parameter specifies the data type to create, the second specifies a *count*, or number of elements, that the type can contain, and the third specifies a *capacity* reserving additional room for the elements. The *capacity* parameter is optional in all cases, and does not limit the bound of the data type. Both the *count* and *capacity* parameters are optional for channel types, which will be introduced later. Container types 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 environment will automatically release memory that is no longer used by the program. Garbage collection and Go-specific collection is discussed in more detail later; however, 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.

2.8 Control Flow

Go provides a variety of mechanisms to alter the control flow of a program. Here we list the standard mechanisms. Section 2.10 discusses a Go-specific excape mechanism.

2.8.1 If Else

The if/else constructs in Go is similar to C's.

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

2.8.2 Switch

Switch statements in Go provide a way to branch across multiple conditions. Unlike C, the case statements in the switch can be expressions. If the case matches or the case-expression is *true* then the body of that case is executed.

```
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 fall through is desired, the programmer must use the *fallthrough* keyword.

2.8.3 Loops

The only looping construct, aside from using *goto* or recursion, is the *for* loop. The *range* keyword simplifies looping from the built-in container types

(slices and maps). *range* produces two values, an index number, and value.

```
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*.

A more traditional loop with a condition can be supplied, as similar to *while* loops in C.

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

As with C, a *for* loop in Go can also define initialization, termination condition and iteration.

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

A *for* without a terminating condition can act as an infinite loop. *break* statements can be issued to exit loops, and *continue* statements can return execution to the beginning of a loop at the next iteration.

2.9 Higher-Order Functions

Functions in Go are first class objects. They can be passed between functions as arguments. In the following 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. The *calculatte* routine is then

```
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)
}
```

called to compute a running sum over the values using the function it was passed for computation, *f*.

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

2.10 Defer

A defer statement can be a function or block of code 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 early. This can be very 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.

In this example, *file.Close()* is registered as a deferred function and 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

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

    defer file.Close()

    if maybeFail() {
        return
    }

    file.Close()
}
```

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 data are evaluated at the time the deferred function is registered.

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

Even though *defer* is called after the last statement in the function, 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.

A function can have multiple deferred statements. The statements are treated as a stack, so the most recently registered flow will be executed first.

The following example illustrates the stack discipline, while also showing that deferred statements can be defined as anonymous higher-order functions.

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

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

2.11 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 judgements, and introduce bugs which compromise the integrity of their programs. Go was designed to eliminate the need for programmers to make difficult concurrency decisions, and to remove as much human-error as possible. The language uses a simple and safe method of computing data concurrently based on C.A.R Hoare’s *Communicating Sequential Processes* (coroutining) concepts [36].

Coroutines, or more aptly called *goroutines* in Go, eliminate concurrent access of data through the inherent design of the language. *channels* and the *go* keyword, permit data to be shared without the need for the programmer to make explicit synchronization calls. This guarantees that multiple con-

current mutating threads never manipulate the data simultaneously, which can compromise the integrity of the data.

Goroutines are functions that execute concurrently with other goroutines (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 goroutines concurrently. If any thread is blocked, the goroutines on another system thread will still continue their execution [23].

The *go* keyword can prefix any function, causing it to run as a light-weight thread in parallel with other threads, including the main thread of execution.

```
func greeting() { println("Hello!") }

func main() {
    go greeting()
}
```

In the example above, the call to *greeting* will be executed in a separate goroutine 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. The following 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

```
func greeting(ch chan int) {
    println("Hello!")
    <- 1
    println("World!")
}

func main() {
    ch := make(chan int)
    go greeting(ch)
    <- ch
}
```

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 goroutine has completed, but it does mean that the routine is in a state whereby it has data ready for any other goroutine listening on the channel. In this example *main* will continue executing as soon as it gets data (which it ignores). *greeting* just sends an arbitrary value down the channel. 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 goroutine 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 goroutine, irrespective of how slow the receiver might be.

2.12 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.

2.12.1 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.

2.12.2 GCC

While not as fast in compile time as the Google compiler, gccgo can produce smaller and highly optimized binaries. The gccgo frontend 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 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.

Chapter 3

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. Since memory is a limited resource, properly managing this is important for developing safe and efficient programs. System crashes can occur if memory is not managed correctly. Improperly managed memory can reduce the amount of memory available for other processes to use. This chapter introduces some basic concepts of memory management and looks at solutions that try to remove the programmer from this complicated task.

3.1 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 compute power and gaining larger main memory stores (increasing RAM size), more processes are being executed on these machines in parallel. With more processes being run, applications need to be written in a way that will not "hog" all of the memory from the simultaneously executing applications. In other words, the fact that a machine has more memory does not permit programmers from managing memory as if it were an unlimited resource.

Application portability is another factor for programmers to consider when writing their programs. Having one application that executes on multiple environments is very useful, as it requires either sharing an existing binary, or simply 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.

3.2 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.

3.2.1 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 harddrive 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 harddisk, the drive must first move its read/write head to a specific location and then transfer that information across a 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.

3.2.2 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 harddrive, 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 solid-state harddrives are becoming more common; however, these disks are still slower to access than RAM, due to data having to traverse a longer bus path to reach the CPU.

3.2.3 Caches

The next tier of memory is the cache. 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 is an important concept which influences system performance via cache behavior. The results of our memory management research does impact cache and overall system performance, therefore having a basic understanding

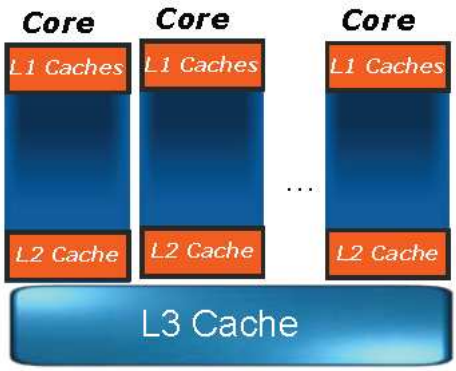


Figure 3.1: Intel CPU caches

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 3.1 illustrates this design.

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 per CPU core and also contains both data and instruction. Even closer to the CPU is the level 1 cache, which is divided into instruction and data. The most recent 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 *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 check the RAM for the data. If the data cannot be found there, it must be obtained from the external store.

3.2.4 Registers

Data must reside directly in the CPU registers to perform computations. 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.

3.3 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 range of addresses 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 3.4.1.

3.3.1 Paging

As previously mentioned, a processes operates within a virtual address space. What happens if a processes 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).

3.4 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.

3.4.1 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, which means that each concurrently running process will

contain 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)[65]. These addresses are virtual. Even though these 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 virtual address of a process 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 3.2 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 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 descriptions define the memory segments of a process, beginning at the *text* segment and working towards the higher addresses.

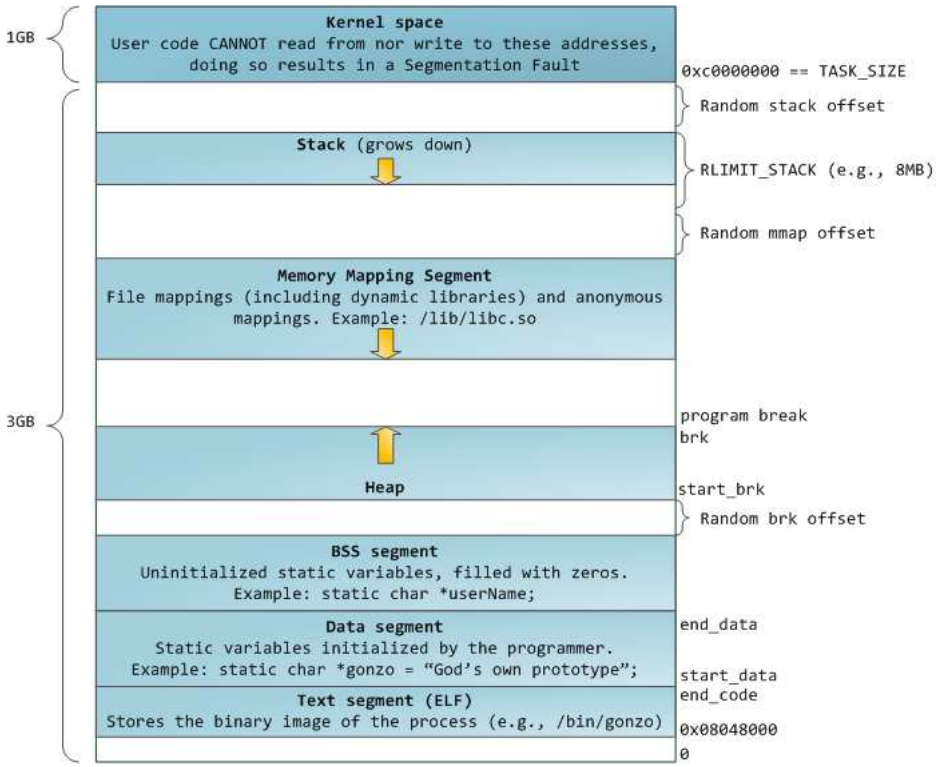


Figure 3.2: Linux process in memory

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 This 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 is produced

from. Typically, this data 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 is temporary and lasts for the duration of the function 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 [65].

3.5 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 make up 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, the data necessary for containing the local variables in each function must be obtained from somewhere. This is the purpose of the call stack, also known as the activation record or stack frame. When each function is called, the stack segment associated to the process is expanded to hold enough data for the locally allocated variables and formal parameters of the function. This stack will grow and shrink depending on the size and number of variables that each function being executed has.

When a function calls another function, the stack frame for the callee will be produced from the 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* 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.

3.6 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[1] in the process' memory space. Memory allocated from this segment is not reclaimed once the function terminates, and can live across multiple function calls, or even throughout the life of

[1]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 program. The latter is often the sign of a memory leak; memory that is allocated but never reclaimed.

Suppose a program takes as input from the user an arbitrary integer value, 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:

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

func buildList(num int) *Node {
    var head *Node

    for i:=0; i<num; i++ {
        elt := new(Node)
        elt.id = i
        elt.next = head
        head = elt
    }

    return head
}
```

The routine above creates a 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 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. Since this example is for Go, the programmer

does not have to worry about memory reclamation, as the garbage collector will reclaim that memory automatically.

When analyzing the lifetime of objects associated to 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 for 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 fully in control of 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 *extern*al functions being called. Dealing with external libraries complicates the programmer's job of writing resource-safe applications.

3.7 Dynamic Allocation Problems

The manual management of memory is a complicated task for programmers. The allocated memory can be alive 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.

3.7.1 Fragmentation and Locality

A memory allocator must return a contiguous chunk of memory to the process, if not, the data for a single object (or array) would be distributed all

throughout the process and access of an object would not produce the proper value. Consider the case of a 32bit 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 a later allocation request. Fragmentation is a property of the underlying allocation routine (*e.g.* malloc) and results in the program using more memory than necessary. For instance, a highly fragmented process will have non-contiguous gaps of *free*d 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 potential for the system to experience multiple cache misses, and subsequent 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.

3.7.2 Memory Leaks

Memory leaks drain a system of its available resources, which can impact the performance of other concurrently executing programs. These leaks commonly occur in languages which require the programmer to manually manage the reclamation of requested/allocated memory. If the memory is never reclaimed, then the memory will remain in use for the lifetime of the program.

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 protect from sharing memory between executables, 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.

3.7.3 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 accessing array data that is outside the bounds of the array. The following Go example illustrates this case.

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

There are two problems here. First, the address containing the value at 1000 integer sizes past the start 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, which might not have even been an integer. 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 to 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.

3.7.4 Dangling Pointers

A dangling pointer is a specific type of invalid memory access. These variables contain an address that *points to* an invalid address. The pointer variable is not problematic, but the data it points to is. Accessing the value pointed at by the the variable is incorrect. Dangling pointers occur when the pointer variable contains an address to dynamically allocated data that has been reclaimed. In such a case the pointer might contain an address to 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. In the cases where the address becomes a value outside of the process space, or references data in a protected memory segment, the OS will issue a segmentation fault. If the fault is not handled properly the program can prematurely terminate.

Dangling pointers occur in languages that require the programmer to manually manage 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 must be carefully designed to prevent these cases [15]. Since

automated memory management systems ensure that all live variables can only reference other live data, they are immune to the problems caused by dangling pointers.

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

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

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

In this example the *n* variable 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 void of the problems caused by *nil* value references. Some languages (such as Go) have a *nil* value which can be used to set/initialize pointer variables to a value of nothing (they point to an address of 0). Since these pointers point to nothing, reading or writing to *nil* is a memory violation. Setting a pointer to *nil* is also a hint to the garbage collector that the data being pointed to is no longer needed.

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:

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

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 reclaim operation, as it is a garbage collected language; however, programmers can influence the results of garbage collection 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.

3.8 Object Lifetimes

A variable's lifetime defines how long a value stays in scope. A variable is said to be *live* at any particular program point if its contents can be accessed. When the contents are no longer accessible, the variable is said to be unreachable or *dead*. Variables declared locally within a function reside on the stack and only have scope within that function. When the function

terminates and the stack frame is popped, those stack-allocated variables for the function are no longer accessible and therefore become *dead*.

Dynamically allocated data have a lifetime that can extend across multiple function calls. 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.

In languages that require the programmer to manually manage memory, the allocated data must be explicitly *free*d 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 determine when the memory can be reclaimed. This reclamation point can occur at a later time of 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 maximal 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.

3.9 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. The following sections provide information regarding solutions to the memory management problems presented earlier.

3.9.1 Region-Based Memory Management

Region-based memory management is a combination of static analysis and runtime instrumentation. 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 the region inference static analysis, the compiler can transform the program by inserting function calls to region operations. Region operations are responsible for managing the creation, removal, and allocation of data from a region. These 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. These operations can either be inserted into the program automatically via the compiler or manually via the programmer.

$CreateRegion()$ Create an empty region from which data structures can be allocated.

$AllocateFromRegion(r, n)$ Allocate n bytes from region r.

$RemoveRegion(r)$ Reclaim all of the memory from region r so that it can be reused later.

RBMM can significantly reduce the execution time it takes 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 individually and then reclaiming its memory.

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

page-in the other objects from that region. This means that the related objects, which also might be accessed at a similar time, will already be in the fast system cache and not also have to be paged-in.

RBMM is not perfect. Since a region is reclaimed all at once, all of the objects in that region must no longer be needed (*e.g.* no other live 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 alive. The RBMM runtime system cannot remove this region until all of its objects are no longer alive. This is what we call *region bloat*. An ideal RBMM system will try to avoid this situation as best it can, to lower the memory pressure of the system.

Automatic and Manual RBMM

An RBMM system can be implemented as either a manual or fully-automatic system. In the former, the programmer is responsible for inserting the region operation calls. 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. Arguably, 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, the compiler determines all of the object relationships and lifetimes, and transforms the program accordingly.

Manual RBMM does not achieve the memory-safety that a fully-automated system can. It relies on the programmer to make decisions about objects in addition to regions. This method suffers the same problems as with traditional manual memory management. A manual RBMM system can try to analyze the programmer's annotations at compile-time and issue compilation errors/warnings if there are any inconsistencies detected, such as removing a region before all of the objects from it are truly dead. A benefit of manual

RBMM is that a good programmer can create regions that do not suffer the *bloat* problem as discussed above.

A fully-automated RBMM system requires no programmer intervention. However, a programmer can influence the compiler to make certain decisions about how regions are managed [62]. 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. Any regions that belong to those objects must also remain alive, causing region bloat. A fully-automated system can excuse the programmer from having to remember when to reclaim an object.

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 [15]. 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 [61, 48, 9].

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 from younger regions to older ones [45]. In such a system, the regions are reclaimed as a stack *pop* operation. This ensures that

a region will never create dangling pointers[15].

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 [19]. Access to a child region can only occur if the parent has unlocked the child.

3.9.2 Garbage Collection

A common system of automatic memory management is garbage collection [39]. 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 computationally expensive than RBMM systems since most of their work is performed at runtime. In contrast to RBMM, GC systems do not suffer the problems of objects that do not have statically-decided lifetimes (e.g. global variables).

Root Set

A GC first begins its operation by scanning a set of addresses which are live in memory. This set, 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 GC will transitively follow 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 to 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.

Conservative and Tyoe-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 lan-

guages, 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 as 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/written to or from by the *mutator* (the non-garbage-collector portion of the program). This is the same issue which complicates concurrent programming. The simplest solution is for the collector to pause program execution (*stop-the-world*), including all threads that might be concurrently executing on behalf of the program, 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 collectors are designed to avoid halting the mutator for significant amounts of time. 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[39]. However, write access to the 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 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 (even if the GC is implemented as stop-the-world), for instance, as a concurrent mark-sweep style collector [13, 41]. 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.

3.10 Object Relationships

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

3.10.1 Object Points-to Relation

The region inference portion of an RBMM analysis can define regions as being sets of objects based on a points-to relationship between these objects. This relationship constructs a set of objects which have a transitive closure of reachability. 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 gaining region bloat, due to a few live objects keeping a region containing many dead objects alive.

This connectivity association has also been utilized in *Connectivity-based* GCs to improve collection time and memory usage [34].

3.10.2 Object Lifetime Relation

To maximize memory utilization, a program must allocate memory at the latest possible time before the object is referenced. The memory must also be reclaimed at the soonest possible time. 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 assure 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; however, object lifetime analysis is a complicated analysis. Past research has shown that object connectivity (points to) is useful in predicting object lifetime and such information can benefit automatic memory management [35].

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 [63, 38, 39, 31]. This hypothesis is not attributed to one person, but is commonly referenced in the garbage collection 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 have been shown to be helpful for reducing the amount of work that a GC has to perform, while also reducing memory footprint. [45].

Chapter 4

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

4.1 Introduction

The garbage collected Go language has features that pose interesting challenges for region inference. While this language does not provide an RBMM system, our work does just that, in hopes of improving time and space performance over that of Go's existing garbage collector. In this chapter we introduce RBMM as an automatic memory management solution that can co-exist with Go's existing garbage collector. Our solution provides a novel design which combines static analysis, to guide region creation, and lightweight 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 source code, making our design more practical than existing RBMM systems.

We have implemented RBMM as an extension to the `gccgo` compiler. Our prototype implementation so far handles almost all of the first order sequential fragment of Go. In fact, 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 give in Figure 4.3. Reflecting the fact that we deal with three-address code, we have normalized the fragment in obvious ways, requiring for example that selectors, indexing, and binary operations are applied to variables, rather than to arbitrary expressions. Note that discussion on the go-routine statements: `go`, `recv`, and `send` appears in Chapter 6.

In Chapter 2 we discussed the syntax and functionality of the Go language. Recall that Go programmers are required to 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.

4.2 Motivation

Different languages pose different challenges for RBMM; for example, logic programming languages require RBMM to work in the presence of backtracking. Here we present our experiences with implementing RBMM for Go. 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). Chapter 6 focuses on how go-routines can be handled via RBMM, this chapter only foucses on the sequential parts of Go. The sequential fragment of Go 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: if a function f calls a deferred function d, execution of d is scheduled to happen just before control

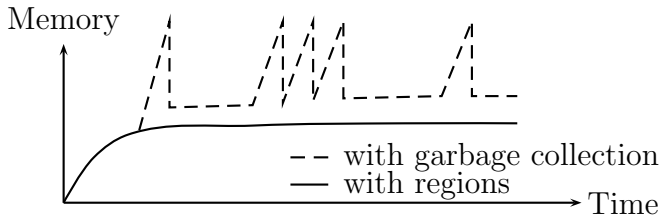


Figure 4.1: Memory occupancy, with ideal program analysis

is handed back to f's caller.

The following two figures (Figures 4.1, 4.2), compare the memory footprint of a hypothetical program using either GC or RBMM. Figure 4.1 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[1] into regions of sufficient size as to not generate region bloat, while also reclaiming memory at the soonest possible time. In contrast, Figure 4.2 looks at a more conservative RBMM analysis resulting in a worse-case scenario for regions, whereby a region becomes increasingly large and acts as a memory leak. The latter is analogous to the region bloat problem, which we discuss in Chapter 3 and in Section 4.3.1.

Our motivation is to compare the performance (both space and time) between Go programs using the existing Go GC system versus our RBMM system. We hope to achieve a more predictable and consistent footprint as presented in Figure 4.1.

[1]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.

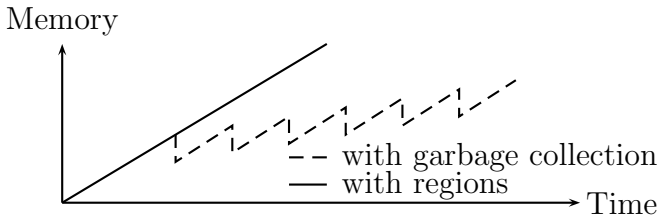


Figure 4.2: Memory occupancy, with imprecise program analysis

4.3 Region-Based Memory Management

RBMM systems must annotate every memory allocation operation with the identity of the region that should supply the memory. The system must also insert, into the program, calls to the functions implementing the primitive operations on regions. 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. 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 freed until the

$$
\begin{array}{lcl}
\mathit{Prog} & \rightarrow & \mathit{Func}^* \\
\mathit{Func} & \rightarrow & \texttt{func}\ \mathit{Fname}\ (\ \mathit{Var}^*\ )\ \{\ \mathit{Stmt}^*\ \texttt{return}\ \mathit{Var}\ \} \\
\mathit{Stmt} & \rightarrow & \mathit{Var} = \mathit{Var} \\
 & | & \mathit{Var} = * \mathit{Var} \\
 & | & * \mathit{Var} = \mathit{Var} \\
 & | & \mathit{Var} = \mathit{Var}\ .\ \mathit{Sel} \\
 & | & \mathit{Var}\ .\ \mathit{Sel} = \mathit{Var} \\
 & | & \mathit{Var} = \mathit{Var}\ [\ \mathit{Var}\ ] \\
 & | & \mathit{Var}\ [\ \mathit{Var}\ ] = \mathit{Var} \\
 & | & \mathit{Var} = \mathit{Const} \\
 & | & \mathit{Var} = \mathit{Var}\ \mathit{Op}\ \mathit{Var} \\
 & | & \mathit{Var} = \texttt{new}\ \mathit{Type} \\
 & | & \mathit{Var} = \mathit{Fname}\ (\ \mathit{Var}^*\ ) \\
 & | & \mathit{Var} = \texttt{go}\ \mathit{Fname}\ (\mathit{Var}^*) \\
 & | & \mathit{Var} = \texttt{recv from}\ \mathit{Var} \\
 & | & \texttt{send}\ \mathit{Var}\ \texttt{on}\ \mathit{Var} \\
 & | & \texttt{if}\ \mathit{Var}\ \{\ \mathit{Stmt}^*\ \}\ \texttt{else}\ \{\ \mathit{Stmt}^*\ \} \\
 & | & \texttt{loop}\ \{\ \mathit{Stmt}^*\ \} \\
 & | & \texttt{break}
\end{array}
$$

Figure 4.3: A representative Go/GIMPLE fragment

region containing item A is freed, 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 freed before any of the others. Since all the pages in those regions would be freed 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 4.4.3. This analysis data is used by the compiler to transform the program by inserting region operations. These calls, or region annotations, are responsible for creating and reclaiming regions, as well as allocating memory from a region

at runtime.

4.3.1 Challenges of RBMM

While RBMM systems have great potential, they also pose significant challenges.

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 4.2 illustrates this kind of memory leak. Each kind of behaviour is observed in practice [53]. 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 too-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 alive, 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 in Chapter 3. This problem does not exist in GC systems.

4.4 Implementation

In this section we discuss our RBMM design for a subset of the Go language. This initial implementation does not handle the parallelization features of Go (*e.g.* go-routines and channels).

4.4.1 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 as opposed to the global region.

4.4.2 Region Instrumentation

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 bigger 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 the pointer on every 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.

A *region* is a linked list of pages. The region header contains bookkeeping information about the region, such as its most recent page. 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 just 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. 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. Reclamation of a region simply means returning its list of pages to the freelist. Quick reclamation is one key benefit of RBMM over that of manual and GC systems.

The following region operations are inserted into the program to imple-

ment RBMM:

- CreateRegion(): Create an empty region from which memory items can be allocated.
- AllocFromRegion(r, n): Allocate n bytes from region r.
- 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.
- 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 4.4.4.
- DecrProtection(r): Decrement the region's protection count.

4.4.3 Program Analysis

The job of our analyses is to decide, for each pointer-valued variable in the program, which region should hold the items to which it points. Our analysis is implemented by two passes: a 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.

Preventing Dangling Pointers

Our analysis 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

$$Map = Func \to EqConstrs$$
$$\mathcal{S} : Stmt \to Map \to EqConstrs$$
$$\mathcal{F} : Func \to Map \to Map$$
$$\mathcal{P} : Prog \to Map$$
$$\mathcal{S}[\![v_1 = v_2]\!]\rho \;=\; (\mathsf{R}(v_1) = \mathsf{R}(v_2))$$
$$\mathcal{S}[\![v_1 = *v_2]\!]\rho \;=\; \mathcal{S}[\![*v_1 = v_2]\!]\rho \;=\; (\mathsf{R}(v_1) = \mathsf{R}(v_2))$$
$$\mathcal{S}[\![v_1 = v_2.s]\!]\rho \;= (\mathsf{R}(v_1) = \mathsf{R}(v_2))$$
$$\mathcal{S}[\![v_1.s = v_2 s]\!]\rho \;= (\mathsf{R}(v_1) = \mathsf{R}(v_2))$$
$$\mathcal{S}[\![v_1 = v_2[v_3]]\!]\rho \;=\; \mathcal{S}[\![v_1[v_3] = v_2]\!]\rho \;=\; (\mathsf{R}(v_1) = \mathsf{R}(v_2))$$
$$\mathcal{S}[\![v = const]\!]\rho \;=\; true$$
$$\mathcal{S}[\![v_1 = v_2 \; op \; v_3]\!]\rho \;=\; true$$
$$\mathcal{S}[\![v = \texttt{new}\ t]\!]\rho \;=\; true$$
$$\mathcal{S}[\![\texttt{if}\ v\ \texttt{then}\ \{\ s_1 \ldots s_n\ \}\ \texttt{else}\ \{\ t_1 \ldots t_m\ \}]\!]\rho$$
$$= (\bigwedge_{i=1}^{n} \mathcal{S}[\![s_i]\!]\rho) \wedge (\bigwedge_{i=1}^{m} \mathcal{S}[\![t_i]\!]\rho)$$
$$\mathcal{S}[\![\texttt{loop}\ \{\ s_1 \ldots s_n\ \}]\!]\rho \;=\; (\bigwedge_{i=1}^{n} \mathcal{S}[\![s_i]\!]\rho)$$
$$\mathcal{S}[\![\texttt{break}]\!]\rho \;=\; true$$
$$\mathcal{S}[\![v_0 = f(v_1 \ldots v_n)]\!]\rho \;=\; \theta(\pi_{f_0 \ldots f_n}(\rho(f)))$$
$$\text{where } \theta = \{f_0 \mapsto v_0, \ldots, f_n \mapsto v_n\}$$
$$\mathcal{F}[\![\texttt{func}\ f\ (f_1 \ldots f_n)\ \{\ s_1 \ldots s_m;\ \texttt{return}\ f_0\ \}]\!]\rho$$
$$= \left[f \mapsto (\bigwedge_{i=1}^{m} \mathcal{S}[\![s_i]\!]\rho)\right]$$
$$\mathcal{P}[\![d_1 \ldots d_n]\!] = \mathit{fix}\left(\bigsqcup_{i=1}^{n} \mathcal{F}[\![d_i]\!]\right)$$

Figure 4.4: Region constraint generation

pointer problem by unifying objects based on a points-to analysis. This analysis forms the basis of our variable unification (equivalence constraint) generation, which infers regions without dependencies between regions.

At compile time our analysis generates object associations based on which objects point to which other objects. Those associated objects are unified by our analyses making them belong to the same region. Therefore, at runtime their corresponding allocations will come from the region they share. In the prior example, object A and B would both be produced from the same region, therefore A cannot be reclaimed before B and vice-versa. Instead, the memory for both A and B will be reclaimed at the same time.

Intraprcedural and Interprocedural Analyses

Our analysis associates with each variable v in the program (or *program variable*) its own *region variable*, which we denote $\mathsf{R}(v)$. If v_1 is a pointer, then $\mathsf{R}(v_1) = r_1$ 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 $\mathsf{R}(v_2) = r_2$ 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 $\mathsf{R}(v_3) = r_3$ means nothing, 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 equivalance 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 righthand side of an assignment with the same set as the lefthand side. For

example, the assignment `a = b` would cause us to generate the constraint $\mathsf{R}(a) = \mathsf{R}(b)$. If the final constraint set built by our analysis does not require $\mathsf{R}(v_1) = \mathsf{R}(v_2)$, 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 4.4 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. $\mathcal{S}$, $\mathcal{F}$, and $\mathcal{P}$ generate constraints for statements, function definitions, and programs respectively. The semantic function $\mathcal{S}$ is used to produce a set of equivalance constraints from a given statement and *Map*. $\mathcal{F}$ is a semantic function producing a new *Map* from a given function and *Map*. $\mathcal{P}$ 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 $\mathsf{R}(v_2)$, the region in which v_1 is stored will be $\mathsf{R}(v_2)$. The region that v_1 *points into*, that is, $\mathsf{R}(v_1)$, can thus be reached from $\mathsf{R}(v_2)$. Most RBMM systems handle such assignments by establishing a dependence between $\mathsf{R}(v_1)$ and $\mathsf{R}(v_2)$ requiring $\mathsf{R}(v_2)$ to be reclaimed before $\mathsf{R}(v_1)$ (if $\mathsf{R}(v_1)$ were reclaimed while $\mathsf{R}(v_2)$ is in use, some pointers in $\mathsf{R}(v_2)$ could be left dangling). This scheme allows, for example, the cons cells of a list to be stored in a different region from the elements of the list. If the cons cells are temporary while the elements are longer lived, this allows the list skeleton to be reclaimed earlier. The implementation discussed in this chapter does not incorporate this refinement; however, the modified RBMM design presented in Chapter 5 does. 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-else`s. In Go/GIMPLE, all loops look like infinite loops with `break` statements inside `if-then-else`s. 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 ρ, 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 called function in ρ (this is what $\rho(f)$ does); this will yield a constraint. It then projects that constraint onto the formal parameters of the callee ($f_1 ... f_n$), including the one representing the return value (f_0). This discards all the primitive constraints involving variables other than formal parameters, but keeps their implications. For example, given the constraints $\mathsf{R}(f_1) = \mathsf{R}(v_5) \wedge \mathsf{R}(v_5) = \mathsf{R}(f_2)$, the projection yields $\mathsf{R}(f_1) = \mathsf{R}(f_2)$. 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 $\mathsf{R}(v_8) = \mathsf{R}(v_9)$.

This process obviously depends on ρ containing the correct constraint for every function in the program. This is determined by the fixed point computation in the definition of $\mathcal{P}$. For $\mathcal{F}$, we begin our analysis with ρ 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 ρ 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 ρ just computed, until the analysis reaches a fied point (when the resulting ρ is the same as it was in the previous iteration).

Figure 4.5 is an example program from which our analysis produces the

following constraints. (Some additional constraints will occur for temporary variables introduced in the GIMPLE code, but we ignore those here):

- `CreateNode`: $\mathsf{R}(\texttt{CreateNode}_0) = \mathsf{R}(\texttt{n})$,
- `BuildList`: $\mathsf{R}(\texttt{n}) = \mathsf{R}(\texttt{BuildList}_1) \wedge \mathsf{R}(\texttt{CreateNode}_0) = \mathsf{R}(\texttt{n})$
- `main`: $\mathsf{R}(\texttt{n}) = \mathsf{R}(\texttt{head})$.

This is inherently a whole-program analysis, and that threatens to make it impractical for real use. Therefore, we have carefully 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 arm of a conditional will be executed, are not significant. This helps make the analysis scalable. More importantly, and contrary to most existing RBMM implementations to date 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.

4.4.4 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, we must also transform functions to take regions as input arguments.

```
1  package main
2  type Node struct {id int; next *Node;}
3
4  func CreateNode(id int) *Node {
5      n := new(Node)
6      n.id = id
7      return n
8  }
9
10 func BuildList(head *Node, num int) {
11     n := head
12     for i:=0; i<num; i++ {
13         n.next = CreateNode(i)
14         n = n.next
15     }
16 }
17
18 func main() {
19     head := new(Node)
20     BuildList(head, 1000)
21     n := head
22     for i:=0; i<1000; i++ {
23         n = n.next
24     }
25 }
```

Figure 4.5: Creating a linked list in Go

```
1  package main
2  type Node struct {id int; next *Node;}
3
4  func CreateNode(id int, reg *Region) *Node {
5      n := AllocFromRegion(reg, sizeof(Node))
6      n.id = id
7      RemoveRegion(reg)
8      return n
9  }
10
11 func BuildList(head *Node, num int, reg *Region)
12     n := head
13     for i:=0; i<num; i++ {
14         IncrProtection(reg)
15         n.next = CreateNode(i, reg)
16         DecrProtection(reg)
17         n = n.next
18     }
19     RemoveRegion(reg)
20 }
21
22 func main() {
23     reg1 := CreateRegion()
24     head := AllocFromRegion(reg1, sizeof(Node))
25     IncrProtection(reg1)
26     BuildList(head, 1000, reg1)
27     DecrProtection(reg1)
28     n := head
29     for i:=0; i<1000; i++ {
30         n = n.next
31     }
32     RemoveRegion(reg1)
33 }
```

Figure 4.6: Same program with region annotations

As discussed in Section 4.4.3, our analysis only summarizes the region equivalance 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 as soon as possible 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 protection 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 4.6 shows the automatically transformed version of the code in Figure 4.5.

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

$$\boxed{Syn_1} \rightsquigarrow \boxed{Syn_2}$$

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 program P. $\mathsf{compress}_f\langle r_0, r_1, \ldots r_n\rangle$ is the list of regions $\langle r_0, \ldots r_n\rangle$, without duplicates, as implied by the region equivalance constraints for f's formal parameters $(f_1, \ldots f_n)$ and return value (f_0). $\mathsf{reg}(f)$ is the set of all distinct regions needed for the definition of function f, as determined by $\mathcal{P}(P)(f)$. $\mathsf{ir}(f)$ is the set of distinct regions of the parameters of function f, that is $\mathsf{ir}(f) = \mathsf{compress}_f\langle \mathsf{R}(f_0), \mathsf{R}(f_1), \ldots \mathsf{R}(f_n)\rangle$. (Since these regions are given to f by its

caller, they are f's *input* regions.) $\mathsf{used}(S_1; \ldots S_n)$ is the set of regions used by any of the statements $S_1; \ldots S_n$. $\mathsf{nonlocal}(S)$ 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 may need to outlive S.

Region-Based Allocation

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

$$\boxed{v \texttt{ := new } (t)} \rightsquigarrow \boxed{v := \mathsf{AllocFromRegion}(\mathsf{R}(v), size(t))}$$

To store objects with undetermined lifetimes, we use the single special *global region*. This region is created upon program initilization and exists for the duration of the program's execution. Any data allocated from the global region is produced and reclaimed via Go's garbage collector.

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:

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

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.

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

$$\boxed{\begin{array}{l}\texttt{func } f(f_1, \ldots f_n)\ \{ \\ \quad S_1; \ldots S_m; \\ \quad \texttt{return } f_0; \\ \}\end{array}} \quad \rightsquigarrow \quad \boxed{\begin{array}{l}\texttt{func } f(f_1, \ldots f_n)\langle r_0, r_1, \ldots r_p\rangle\ \{ \\ \quad S_1; \ldots S_m; \\ \quad \texttt{return } f_0; \\ \}\end{array}}$$

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

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:

$$\boxed{v = f(v_1, \ldots v_n)} \quad \rightsquigarrow \quad \boxed{v = f(v_1, \ldots v_n)\langle r_0, r_1, \ldots r_p\rangle}$$

$$\text{where } \langle r_1, \ldots r_p\rangle = \mathsf{compress}_f\langle \mathsf{R}(v_0), \mathsf{R}(v_1), \ldots, \mathsf{R}(v_n)\rangle$$

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

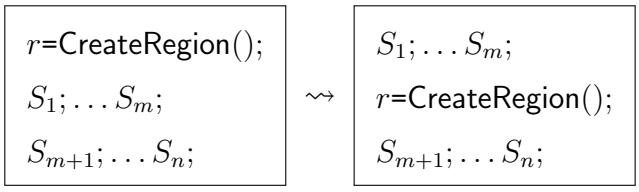
Region Creation and Removal

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 finish with 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.

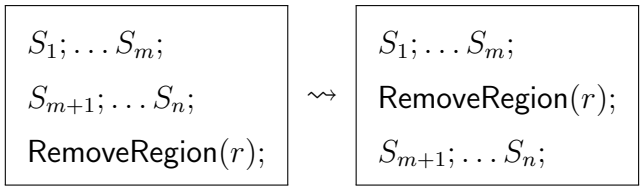
$$\boxed{\begin{array}{l}\texttt{func } f(f_1, \ldots f_n)\ \{ \\ \quad S_1; \ldots S_m; \\ \quad \texttt{return } f_0; \\ \}\end{array}} \quad \rightsquigarrow \quad \boxed{\begin{array}{l}\texttt{func } f(f_1, \ldots f_n)\ \{ \\ \quad C; S_1; \ldots S_m; R; \\ \quad \texttt{return } f_0; \\ \}\end{array}}$$

$$\text{where } C = \{r\text{=}\mathsf{CreateRegion}();\ \mid r \in \mathsf{reg}(f) \setminus \mathsf{ir}(f)\}$$
$$R = \{\mathsf{RemoveRegion}(r);\ \mid r \in \mathsf{reg}(f) \setminus \{\mathsf{R}(f_0)\}\}$$

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.)



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



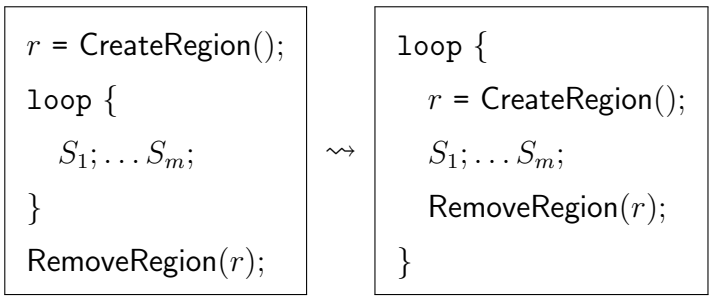
where $r \notin \mathsf{used}(S_{m+1}; \ldots S_n)$

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.

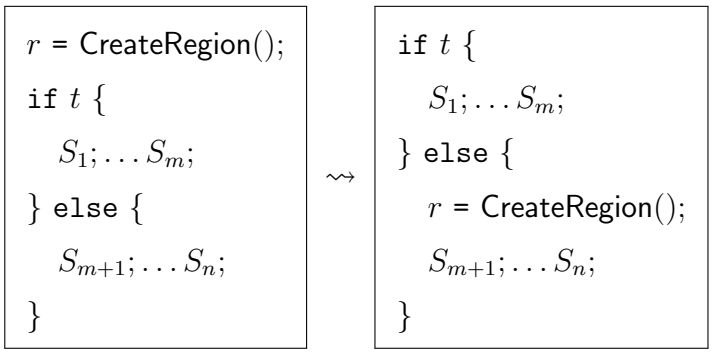
Our current system does move the region removal calls if they are inside a loop. If the creation is outside of a loop, then the removal is moved just outside of the loop. This prevents the dangling case where the loop will reuse a region that was removed in a previous iteration. If the region creation is inside a loop, then the removal is placed as the final statement in the basic

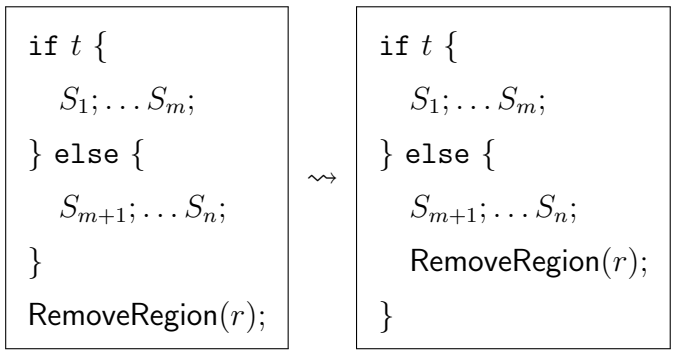
block before the loop jumps back to the loop start. This allows the memory to be reclaimed per loop iteration.



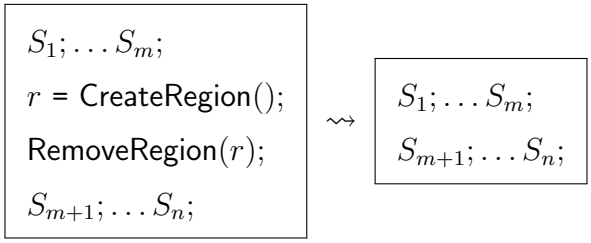
where $r \notin \mathsf{nonlocal}(\texttt{loop}\ \{S_1; \ldots S_m;\})$

The remaining creation/removal optimizations in this section are not implemented but we discuss these here, as they can be helpful for future implmentations. The following transformation may be useful when only one arm of a conditional uses a region:





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



Region Protection Counting

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.

We have not yet implemented a global analysis. Our current 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 $\mathsf{R}(v) = r$ is needed after the call, we invoke $\mathsf{IncrProtection}(r)$ before the call, and we invoke $\mathsf{DecrProtection}(r)$ after the call:

$$\boxed{\begin{array}{l} S_1; \ldots S_m; \\ v = f(\ldots)\langle \ldots, r, \ldots \rangle \\ S_{m+1}; \ldots S_n; \end{array}} \quad \rightsquigarrow \quad \boxed{\begin{array}{l} S_1; \ldots S_m; \\ \mathsf{IncrProtection}(r); \\ v = f(\ldots)\langle \ldots, r, \ldots \rangle \\ \mathsf{DecrProtection}(r); \\ S_{m+1}; \ldots S_n; \end{array}}$$

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

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.

A simple additional transformation can remove unnecessary calls to $\mathsf{IncrProtection}(r)$ and $\mathsf{DecrProtection}(r)$, leaving only the first increment and last decrement.

$$\boxed{\begin{array}{l} S_1; \ldots S_m; \\ \mathsf{DecrProtection}(r); \\ S_{m+1}; \ldots S_n; \\ \mathsf{IncrProtection}(r); \\ S_{n+1}; \ldots S_q; \end{array}} \quad \rightsquigarrow \quad \boxed{\begin{array}{l} S_1; \ldots S_m; \\ S_{m+1}; \ldots S_n; \\ S_{n+1}; \ldots S_q; \end{array}}$$

We have not yet implemented this transformation.

While we have not done so, it should be possible to implement an extra 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 *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 *reference* counts, which count the number of individual pointers to an item or region. For example, in RC [18], 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.

4.4.5 Higher-Order Functions

Go supports the passing of functions as arguments to other functions. This higher-order feature forms the basis of Go's ability to handle function pointers, go-routines, anonymous functions, and deferred functions. 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 determine what regions should be passed in the transformed function call. For instance, consider a program which has many functions with one input argument 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:

$$\boxed{f(e_1)} \rightsquigarrow \boxed{f(e_1)\langle r_1 \rangle}$$

In this example, the signature remains the same on the unmodified functions:

$$\boxed{f(e_1)} \rightsquigarrow \boxed{f(e_1)\langle\rangle}$$

In other words, some functions might be unaltered and still have a single input argument, while other functions might have two. This means that all single pointer argument function pointers do not match those of the transformed functions.

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 requires region arguments, then we insert a *trampoline* function at the assignment site instead of assigning the original function. 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 function pointer call. This occurs for both function call arguments and for assignments to function pointer variables:

$$\boxed{g := f(e_1)} \rightsquigarrow \boxed{g := tr\langle r_1 \rangle\{f\}}$$

The trampoline, tr, takes exactly one region variable argument for each argument the original function 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 function 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 function that would have been assigned, and maps the input arguments of the trampoline to the input arguments of the original function.

4.4.6 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 frontend through the use of an interface method table. Each instance of an object, which belongs to a 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.

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, 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.

4.4.7 Returning Local Variables

The Go frontend handles the return of local variables by allocating memory dynamically and returning the address safely. Since the frontend 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.

4.4.8 Region Merging

There are cases during both compile time and runtime where two regions become associated with one another, such as when a structure is allocated data from one region, and one of its members is allocated data from another region. In this case, a region removal of one region before the other can cause an invalid memory access. If the region containing the member is removed, but its parent is still alive, then access through the parent to the member is invalid. To avoid such situations the related regions can be merged; creating one region from multiple. The primary drawback of region merging is that more data is associated with the region, therefore some objects will be forced to live longer than they might have in a region with fewer objects.

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

$$\boxed{f(e_1, e_2)} \rightsquigarrow \boxed{f(e_1, e_2)\langle r_{1,2}\rangle}$$

Runtime merges occur in two cases. The first case is when our analysis detects at compile time where a merge might occur. This occurs when a merge happens in a branching statement. By inserting a routine to merge data at runtime, based on a conditional, we can reduce the probability that a large region will be generated. 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.

The other runtime merging case occurs when we pass around function pointer trampolines for supporting higher-order functions. The trampoline requires all arguments and their associated regions to be passed as input. Recall that this trampoline wraps the function to be called, and is responsible for calling the function with the proper arguments. As above, if this wrapped-function assumes multiple arguments are to be from the same region, a merge must occur at runtime. Our static analysis cannot determine if the two arguments passed to the trampoline and then to the wrapped-function will be from the same region or not. Therefore, a runtime region merge must be attempted. Our transformation inserts code into the trampoline function before the wrapped-function is called.

The primary 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 is bad as 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.

Higher-order functions require dynamic runtime merging. For instance, the following example will require that $\mathsf{R}(a) \equiv \mathsf{R}(b)$.

In the above example *fn* might point to *foo* where it will require that $\mathsf{R}(a) \equiv \mathsf{R}(b)$. For other function pointers passed as an argument, it is possible that $\mathsf{R}(a) \neq \mathsf{R}(b)$ and that sound. In this case we must dynamically merge $\mathsf{R}(a)$ and $\mathsf{R}(b)$ at runtime if *fn* is a pointer to *foo*. If we were not to do this, a dangling situation could arise if $\mathsf{R}(b)$ were to be removed before $\mathsf{R}(a)$.

```
func foo(a, b *Thing) {
    a.next = b
}

func doStuff(fn func(*Thing, *Thing)) {
    a := new(Thing)
    b := new(Thing)

    // 'fn' is a function pointer,
    // in this case assume it is to 'foo()'
    fn(a, b)

    // 'a' is no longer needed, but 'b' should remain
    bar(b)
}
```

4.4.9 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 the only access to the field might be from the RBMM allocated structure which is outside of the memory space of Go's GC. Therefore, Go's GC will never be able to reach the allocated field from a root-set variable during a memory scan, and incorrectly think 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 library's since we cannot

transform or specialize any functions in these pre-compiled binaries.

Consider the case when a function pointer variable 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 function or datatype. Such cases would require that the functions in the external library file to know how to pass region data to the function pointer input argument, 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 function that the function pointer refers to. The specialized copy 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 function pointer as mentioned in Section 4.4.5.

4.4.10 Go Infrastructure

Go programs can be made up of multiple object files. These object files form the basis of libraries, or packages, for Go. As previously mentioned, 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. 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

Benchmark			GC			RBMM		
Name	LOC	Repeat	Alloc	Mem	Collections	Regions	Alloc%	Mem%
`binary-tree-freelist`	84	1	270	227M	3	1	0%	0%
`gocask`	110	100k	56M	3.8G	97k	700,001	0.5%	0.1%
`password_hash`	47	1k	160M	13G	145k	5,001	˜0%	˜0%
`pbkdf2`	95	1k	115M	8G	92k	12,001	0%	0%
`blas_d`	336	10k	6M	890M	11k	57,0001	9.2%	9.1%
`blas_s`	374	100	49K	5M	58	5,001	10.1%	21.0%
`binary-tree`	52	1	607M	19G	282	2,796,195	˜100%	˜100%
`matmul_v1`	55	1	6K	72M	10	4	96.0%	99.9%
`meteor-contest`	482	1k	3M	165M	2k	3,459,001	˜100%	99.9%
`sudoku_v1`	149	1	40K	12M	110	40,003	98.8%	99.2%

Table 4.1: Information about our benchmark programs

file.

4.5 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 4.1 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 testsuite and aimed at 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" [44], and the remaining programs are from libraries: Michal Derkacz's `blas_d` and `blas_s` [12], Dmitry

Benchmark	MaxRSS (megabytes)			Time (secs)		
Name	GC	RBMM		GC	RBMM	
`binary-tree-freelist`	891.84	892.01	(100.0%)	12.4	12.2	(98.4%)
`gocask`	27.45	27.63	(100.7%)	71.6	69.7	(97.3%)
`password_hash`	26.60	26.80	(100.7%)	119.0	119.1	(100.1%)
`pbkdf2`	26.37	26.58	(100.8%)	71.4	71.6	(100.3%)
`blas_d`	25.87	26.14	(101.0%)	5.4	5.4	(100.0%)
`blas_s`	26.05	26.29	(100.9%)	12.2	12.1	(99.2%)
`binary-tree`	1323.74	1196.51	(90.4%)	79.2	14.7	(18.6%)
`matmul_v1`	313.03	307.87	(98.4%)	11.7	11.7	(100.0%)
`meteor-contest`	27.41	27.11	(98.9%)	11.0	11.0	(100.0%)
`sudoku_v1`	26.96	26.65	(98.8%)	15.6	16.5	(105.8%)

Table 4.2: Benchmark results

Chestnykh's `passwordhash` and `pbkdf2` [8], and Andre Moraes' `gocask` [47]. 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 our analysis infers for a single run 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 4.2 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 labelled GC) and when compiled with our experimental RBMM system (the columns labelled 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 in Ubuntu's libgo0 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 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 4.4.4 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 increased code size and additional internal fragmentation. For one of these benchmarks, `binary-tree`, the saving is pretty 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 live, 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, `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, 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 factor of infinity.

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.

4.6 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 system propagates 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. Chapter 5 extends the concepts presented in this chapter.

Chapter 5

Enhancing Our Go RBMM System

But wait! There's more!

Billy Mays

5.1 Introduction

In this chapter we introduce enhancements to our RBMM system defined in Chapter 4. First, we modify our unification algorithm to allow a single object to belong to multiple regions, if such a structure contains pointers to other data types. Secondly, we try to improve our memory footprint by solving the memory bloat and infinite (global) region problem. For the latter problems, we introduce a garbage collector, which is capable of reclaiming unreachable (dead) objects from regions. Ultimately, this combination of RBMM and GC tries to achieve the advantages of both systems while avoiding the disadvantages. Lastly, we introduce a novel means of reclaiming individual elements from array segments whose elements are no longer reachable.

By combining RBMM and GC we present three contributions, which make our work novel:

- We propose a *new way of combining GC and RBMM*, with *less overhead* than similar systems [26].

- 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 live and others are not. We show how to recover the memory occupied by the dead elements, at a low cost.

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.

5.2 Enhancing Our Existing Analysis

Our initial RBMM system utilized a unification method that placed all items into the same region if there existed a points-to association between them. This means that any item that contained a pointer as a field, would also share the same region with the field. While this 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 structures 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 modified two rules in our semantics as presented in Chapter 4. Originally, the rules were defined as in Figure 5.1. Our modified rules listed seen in Figure 5.2.

As you can see, this rule change is rather simple; however, the cost of implementing this modification was high (it took a long time). For instance, functions no longer require a single region per argument, instead, they now

$$\mathcal{S}[\![v_1 = v_2.s]\!]\rho \ = (\mathsf{R}(v_1) = \mathsf{R}(v_2))$$
$$\mathcal{S}[\![v_1.s = v_2]\!]\rho \ = (\mathsf{R}(v_1) = \mathsf{R}(v_2))$$

Figure 5.1: Original constraint rules

$$\mathcal{S}[\![v_1 = v_2.s]\!]\rho \ = (\mathsf{R}(v_1) = \mathsf{R}(v_2.s))$$
$$\mathcal{S}[\![v_1.s = v_2]\!]\rho \ = (\mathsf{R}(v_1.s) = \mathsf{R}(v_2))$$

Figure 5.2: Modified constraint rules

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 4.4.5 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 our performance. Additionally complicating matters is the case when a non-global and the global region could become merged. This is dangerous, as that means all data from the non-global region would never be reclaimed, not even by Go's existing garbage collector. The latter occurs because the memory was allocated from our RBMM allocator and when RBMM is not used to manage it (*e.g.* the allocations are merged into and belong to the global region), Go's garbage collector has no concept of these addresses, thus it cannot reclaim our RBMM allocated objects. Therefore, runtime merging can effectively introduce memory leaks into a program.

To simplify and eliminate the latter problems we relaxed our semantics such that all variables that are passed as arguments to a function pointer belong to the global region. This is overly conservative but means that Go's garbage collector can be used to allocate and manage their memory (reducing

memory leaks). This 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, therefore we can now collect data from the global region, and not rely on Go's existing collector for such matters. Figure 5.3 describes this change in terms of our semantics.

$$\mathcal{S}[\![v_0 = fnp(v_1 \dots v_n)]\!]\rho \ = \bigwedge_{i=0}^{n} (\mathsf{R}(v_i) = \mathcal{G})$$

Figure 5.3: Added constraint rule for function pointers

5.3 Combining Regions and Garbage Collection

Automated memory management systems can be implemented using either RBMM or GC. A comparative analysis of the two approaches is complex, but the big picture is that there is a time/space trade-off between them. 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 3.

But, 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 simply 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 behaviours that favor GC.

Because the lifetime of a memory item is in general undecidable, region analysis must conservatively approximate it. 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 [25, 5] 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 4 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 we believe may be important. 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 if we cannot merge the two regions at runtime, and indeed that is the case when one of the "regions" is managed by Go's builtin GC system.

The goal of the work presented in this chapter is to create an RBMM system that allows 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 item 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 previous RBMM system (Chapter 4). The existence of the GC system

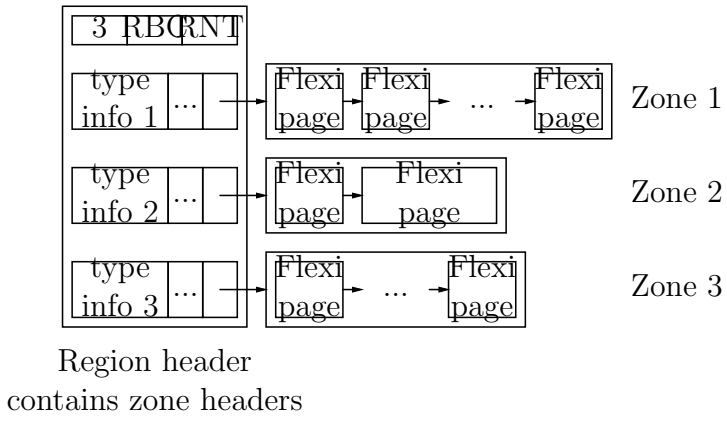


Figure 5.4: Region data structure

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, to improve locality of reference and to reduce internal fragmentation.

Previously all flexipage headers contained a pointer to the next free byte. This pointer is redundant and is not necessary. In fact, our system only needs one pointer to the free space per region. Therefore, another 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.

5.3.1 Managing Ordinary Structures

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 [26]. 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. 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 amount of memory that this scheme allows us to save 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 * 8 = 2.4 MB, which in this case represents 50% overhead. On other cases, the percentage will be different. However, it should be clear that our scheme saves not just significant amounts of memory, but also the time needed to fill in this memory.

While GC systems have used type-specific zones before, to the best of our knowledge, this is the first time they have been applied to regions in a system using RBMM.

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. Until Section 5.5, we will assume that such constructs 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 a description of the type of the items in the zone, what 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 builtin (*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 ike 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 free'd 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 4, with the exception that now there is a zone for each type that can appear in the region. 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.
- Two bits that are needed only during collections. 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 live 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.
- A pointer to the next region. This helps our runtime maintain a list of free regions that can be reused.
- A size which reflects how big the region and its array of zone pointers is. This, as with the previous bullet above, helps our runtime manage the list of free regions that can be reused when a CreateRegion()operation is called.

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.

Most RBMM systems put the region header at the start of the first flexipage of the region. 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 (Pool 2) that is *separate* from the pool that supplies the flexipages for zones (Pool 1).

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 or from a freepage in our allocator's free page list (freelist).

The zone most 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 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 illustrates how our new CreateRegion()function looks:

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

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 since it creates many regions with unused zones; however, this is trivial to implement and will allow us to test the combination of RBMM and GC.

Reclaiming a Region

Reclaiming a region is simple: we release every flexipage in every one of the region's zones back to Pool 1, and we release the region header back to Pool 2.

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 *zone-finder*, which is a variant of the *BIBOP* or *big bag of pages* idea [55]:

- 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, Pool 1, the pool from which flexipages are allocated, is a contiguous sequence of pages.
- We pair every page in Pool 1 with a *shadow* word in a new pool, Pool 3, which is the zone-finder.

- If a page in Pool 1 is not currently in use, then the shadow word corresponding to it will be NULL.
- If a specific page in Pool 1 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 Pool 1 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 always stored at aligned addresses, we use the least significant bit as a tag to distinguish between the last two cases.

Conceptually, Pool 1 and Pool 3 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 Pool 1 as a sequence of pages, and Pool 3 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.

Managing Redirections

We garbage collect each zone using a semispace algorithm [16]; that is, we copy every live 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 live items arrives at an item, it needs to know whether that item has been copied to the to-space yet. (Copying a live 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:

$$n = \left\lfloor \frac{(bytes\_per\_flexipage - bytes\_per\_header) * 8}{1 + (bytes\_per\_item * 8)} \right\rfloor$$

$$bi = \left\lceil \frac{bytes\_per\_header + \lceil \frac{n}{8} \rceil}{alignment} \right\rceil * alignment$$

Algorithm 1 Preserve data in an item

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

```
function Preserve(base, type, fpp, zhp)
    size ← SizeOf(type)
    newbase ← AllocFrom(ToSpace(zhp), size)
    CopyMemory(newbase, base, size)
    RedirectBit(fpp, base) ← True
    *base ← newbase                          ▷ Set redirect pointer
    return newbase
```

where all items begin at an address divisible by *alignment*.

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

Between two garbage collection cycles, each RedirectBit in each flexipage contains a value of zero. When the traversal encounters a live item whose RedirectBit is zero, it copies the item to the to-space, and sets its RedirectBit to a value of one. To let later parts of the traversal know not just that the item *has* been copied but also *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 ok 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 malloc.

When a GC cycle is complete, all the pages of all the flexipages of the collected regions are returned to the freelist of Pool 1. Before any flexipage is reused, all its bits will be set to zero, including its redirection bits.

5.3.2 Collecting Garbage

The top level of our GC algorithm is shown in Algorithm 2. Its first parameter 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-redirected 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 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 live items in regions which *are* in *GC_regions*, but

- the collector will not need space to store copies of all the live 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 live 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.

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

Together these algorithms preserve each live 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/free'd by returning them to the freepage list so that they can be reused for other zones if necessary.

Algroithm 3 locates live items in the regions being collected and copies them to the to-space of their zone. The 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 5.3.1, 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* of the item). If s is the size of the items in the flexipage, then

$$base = fpp + bi + s * \left\lfloor \frac{addr - (fpp + bi)}{s} \right\rfloor$$

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 the regions being collected cannot 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 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 live 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

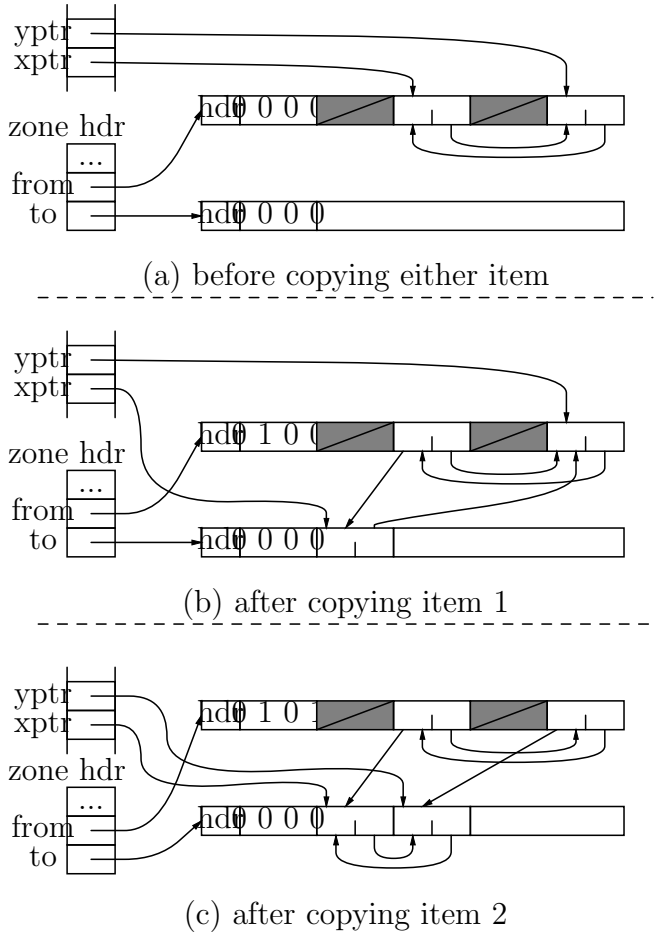


Figure 5.5: Example: copying two items to to-space

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 this 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 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 fragmentations. The garbage that existed in the from-space flexipage has been eliminated and only live data exist contigiously in the to-space.

5.4 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 5.4.2, we treat them as such.

5.4.1 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

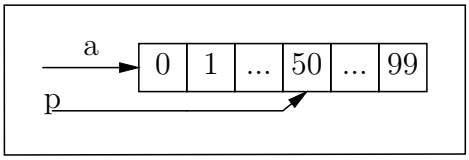
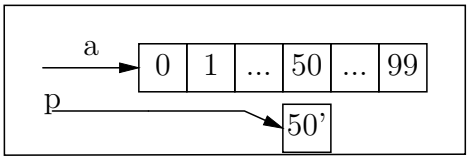


Figure 5.6: Pointer into the middle of an array



New Sub-region Memory Area

Figure 5.7: Duplicate data after collection

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 calculation. 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.

5.4.2 Preserving Only the Used Parts of Arrays

Our PreserveAndTrace algorithm treats any reference to any part of an item as a reference to the entire item. A live 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 live 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 live array elements individually; we must ensure that contiguous sequences of live 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 live. Unfortunately, we cannot know at that time whether the elements before and after these in the array are live 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 live and which are dead only once a GC has finished tracing all live 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 live 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 5.4.1. 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 live 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 has been seen live 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 executables 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

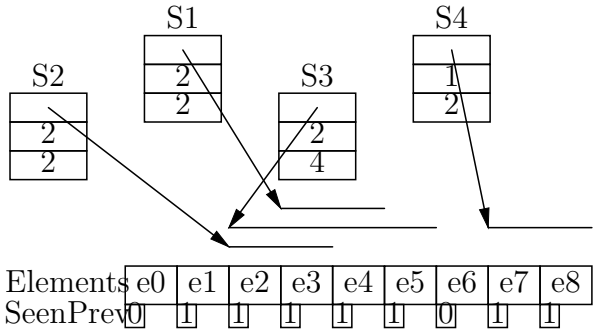


Figure 5.8: Copying only the previously live parts of arrays

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 5.4.1 (represented by function 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 *fse*, the index of the first slice element in the array. From that and the capacity of the slice, we can calculate *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 *fse* to 3 and *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 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 live at 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 live.

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 live 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 Algorithm 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 add 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 5.4.1.

5.5 Handling Other Go Constructs

Go provides several features we have not yet discussed. While we have not modified our collector to handle them thus far, 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 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 zone 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.

Go's co-routines (*goroutines*) present more of a problem. The static analyses used to control the RBMM system assume that a called function runs to completion before the next function is called. The `go` construct violates that assumption: the function call following the `go` keyword will typically not complete before the start of the execution of the following construct. This means in some cases, it is necessary to decide *dynamically* when to reclaim a region. Go also supports *channels* for communication between goroutines. These must be augmented to pass regions along with the items they contain, so that the receiving goroutine can know which regions to allocate from and which to reclaim. Finally, during GC, any thread that may use or modify any items in any regions being collected must be paused for the duration of the GC. Note, however, that threads that cannot access any regions being collected *may* be allowed to continue during GC.

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 paper need minor adaptations to handle these representations.

5.6 Evaluation

In order for us to measure how our RBMM system performs with our region-aware garbage collector, we executed a series of tests to measure time and memory usage on a series of benchmarks. Most of the benchmarks are derivatives of those mentioned in Chapter 4 with a few parameters adjusted in order to increase runtime. Due to complications of our 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 were 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. We only show results for those programs 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 calculator).

These tests were conducted on the same machine as our original series of evaluations (see Chapter 4). 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 12.10. Similarly, the GCC used to compile both our modifications (our plug-in and runtime) 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 that 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.

5.6.1 Methodology

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

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

5.6.2 Results

Table 5.1 contains our benchmarking results. Of interest are both the time and space usage of the benchmarks. The unmodified tests are labeled as *(plain)*. The tests using RBMM with Go's garbage collector are labeled as *(rbmm)*, and the tests utilizing RBMM and our region-aware garbage collector are labeled as *(rbmm+gc)*.

An interesting measure to consider when reading these values 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 process allocation types and to traverse the call stack. Both of our RBMM systems create regions dynamically at runtime. 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 garbage collection, therefore in the test *rbmm* tests, each region contains just a single type agnostic zone.

Table 5.2 illustrates just the binary sizes alone 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	HWM(KB)	Elapsed (seconds)	Inuse Pages		Free Pages	
			N	Bytes	N	Bytes
binary-tree (plain)	91504.80	11.71	NA	NA	NA	NA
binary-tree (rbmm)	1443682.00	3.45	9309	38129664	6205	25415680
binary-tree (rbmm+gc)	1459476.40	8.30	9420	38584320	6965	28528640
fannkuch (plain)	8567.60	7.46	NA	NA	NA	NA
fannkuch (rbmm)	8592.00	7.59	4	16384	1	4096
fannkuch (rbmm+gc)	8631.20	7.37	6	24576	1	4096
mandelbrot (plain)	8636.00	4.49	NA	NA	NA	NA
mandelbrot (rbmm)	8670.40	4.49	0	0	0	0
mandelbrot (rbmm+gc)	8695.20	4.49	0	0	0	0
matmul (plain)	86977.20	13.92	NA	NA	NA	NA
matmul (rbmm)	80622.80	11.82	6005	73830400	4503	55369728
matmul (rbmm+gc)	117000.00	13.19	9011	110813184	7	172032
meteor-contest (plain)	8807.60	0.18	NA	NA	NA	NA
meteor-contest (rbmm)	12750.80	0.12	3	12288	1	4096
meteor-contest (rbmm+gc)	12820.00	0.12	5	20480	1	4096
pidigits (plain)	9760.40	43.01	NA	NA	NA	NA
pidigits (rbmm)	9812.00	42.94	0	0	0	0
pidigits (rbmm+gc)	9830.80	43.05	0	0	0	0

Table 5.1: Early implementation performance results

test is initially loaded into main memory. This size will set a minimum bound on the HWM values depicted in the former table. RBMM alone contributes an average of 97.16 KB to the executable size. Adding in our region-aware GC contributes, on average, an additional 68.16 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 and time constraints. Also note that we have yet to begin optimizing our garbage collector.

It is important to note that our system never returns memory to the OS, instead the system recycles it 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 run-

Test	Executable Size (KB)
binary-tree (plain)	66
binary-tree (rbmm)	146
binary-tree (rbmm+gc)	211
fannkuch (plain)	65
fannkuch (rbmm)	145
fannkuch (rbmm+gc)	204
mandelbrot (plain)	65
mandelbrot (rbmm)	143
mandelbrot (rbmm+gc)	207
matmul (plain)	66
matmul (rbmm)	145
matmul (rbmm+gc)	214
meteor-contest (plain)	78
meteor-contest (rbmm)	157
meteor-contest (rbmm+gc)	247
pidigits (plain)	65
pidigits (rbmm)	144
pidigits (rbmm+gc)	206

Table 5.2: Binary Sizes of the Benchmarks

times (*rbmm* or *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`). The latter is less than 80 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 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.

Test	Go's GC	Our GC (RBMM+GC)
binary-tree (plain)	202	NA
binary-tree (rbmm)	1	NA
binary-tree (rbmm+gc)	0	18
fannkuch (plain)	1	NA
fannkuch (rbmm)	1	NA
fannkuch (rbmm+gc)	0	2
mandelbrot (plain)	1	NA
mandelbrot (rbmm)	1	NA
mandelbrot (rbmm+gc)	1	1
matmul (plain)	11	NA
matmul (rbmm)	1	NA
matmul (rbmm+gc)	0	14
meteor-contest (plain)	7	NA
meteor-contest (rbmm)	1	NA
meteor-contest (rbmm+gc)	0	9
pidigits (plain)	4487	NA
pidigits (rbmm)	4479	NA
pidigits (rbmm+gc)	4459	0

Table 5.3: Number of Garbage Collections

(This is the reason why we cannot fully disable the need for Go's existing garbage collector.) In other cases (`binary-tree` and `meteor-contest`), where dynamic memory is more stressed, both the *rbmm* and *rbmm+gc* versions allocated more memory than the unmodified test. The exception is `matmul` which our *rbmm* version performs better in comparison to the unmodified version. However, this case also shows us that our garbage collector needs work, as it allocates an additional magnitude of order of memory over that of the *rbmm* and *plain* versions. We note that `matmul` creates many slices and our collector only preserves entire (and not portions of) slices. Further, our garbage collector's slice handling has been riddled with bugs and much time has been spent trying to smooth things out, with little to no avail. In fact, given certain input this test will premature exit in failure. The results

of this test compiled here are all from runs which did not terminate early. We also find that `matmul` has significantly fewer free pages during execution, which suggests that our collector might be inefficiently handling pages. As expected, our collector does require more pages than our *rbmm* system along. Recall that a region will need to allocate more pages during copy-collection for copying objects from *from-space* into *to-space*. In contrast, `binary-tree`, which also has a significant negative memory performance (and allocates no slices) shows that our use of pages is efficient in comparison between our modified versions. Both of our modified tests utilize nearly 16 times more memory than the unmodified version, and this finding suggests that our regions must be contributing a fair amount of bloat and that potentially our GC is of little use. These results suggest that we should investigate more at how our slices are managed as well as how effective are collector is performing when collecting regions. We did notice that disabling our slice collection in the *matmul (rbmm+gc)* case did bring down the footprint to a size more representative of the *rbmm* version. However, given our poor performance for *binary-tree* which uses no slices, our memory problems might also be plagued with a problem not solely related to our garbage collector.

Both of our modified systems perform well for execution time on a majority of the tests we examined. Time is dependent on both application runtime and GC performance. Both GC systems, our GC and Go's existing GC, are stop-the world, and halt execution of the mutator to perform their GC function. In addition, our RBMM systems additionally affect runtime performance due to our static analysis inserting region operations during compilation (in the *rbmm* and *rbmm+gc* test cases). Of course these functions are only useful for RBMM and not GC, therefore they provide additional cycles that our RBMM systems must use. The results show that our region-aware GC (*rbmm+gc*) did appear to slow down runtime from that of the *rbmm* tests. Since these differences are often quite minor and the times so small, we cannot exclude the possibility of OS noise and overhead from other con-

currently running processes on the test machine. As with the results seen in Chapter 4, our `binary-tree` performance did quite well over that of the unmodified version. One such cause might be that we collect less frequently than Go's collector. However, it is beneficial to know that the collector does not seem to pose significant time overhead for the low-memory tests.

Table 5.3 shows the number of garbage collections that each test performed. 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 desireable, 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 apparant 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 random or time Go packages. It turns out that the Go runtime 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 permiting 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 or RBMM efficiency, or possibly some combination of both. However, we report just the value from one run from the default Go execution, as we think that such a number will allow us to determine where our speed up is coming from. The values for our region-aware collector are only available on the tests that have our collector enabled *rbmm+gc*, all other cases have results labeled as *NA*.

For the `binary-tree` case, which we perform incredibly well on with re-

spect to time, we see that our collector does execute 18 times, versus the 202 that Go's collector alone performs. We note that a majority of the speedup in the latter is due to less GC. We should also consider tweaking our GC parameters which can increase collection frequency and potentially reduce the HWM for this test as seen in Table 5.1. On the opposite end of the performance table is `pidigits`. 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 collector and RBMM system are fully utilized. While both of our systems perform well with respect to time, our *rbmm* alone system does slightly better for space. In this 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. In the case where we rely on our region-aware GC, we notice that we introduce an incredible amount of space. At this time we are not sure where this additional overhead is originating from. However, even though we do perform 18 GCs we did not compromise much time with respect to the *rbmm* alone case, and we perform better over that of the base *plain* case.

5.7 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 [26], we attach them to pages, similar to the *Big Bag of Pages* approach to GC. The system can decide which region or regions to garbage collect based on runtime memory usage. While not fully implemented, our system 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.

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 method reclaims unneeded elements over two successive GCs, with each GC retaining the elements that were live 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.

Algorithm 3 Preserve an item and everything it keeps alive

Require: *addrptr*: Pointer to the address of an item
Require: *toplevel*: Is the call coming from Algorithm 2?

```
function PreserveAndTrace(addrptr, toplevel)
    addr ← *addrptr
    if addr is in the heap then
        ⟨fpp, base, zhp⟩ ← LookupHeap(addr)
        type ← TypeIn(zhp)
        offset ← addr − base
        rhp ← ContainingRegion(zhp)
        if ¬RegionBeingCollected(rhp) then
            newbase ← base                          ▷ Item is not moved
            needstrace ← RegionNeedsTracing(rhp)
        else
            if ¬RedirectBit(fpp, base) then
                newbase ← Preserve(base, type, fpp, zhp)
                *addrptr ← newbase + offset
                needstrace ← RegionNeedsTracing(rhp)
            else
                newbase ← *base                     ▷ Get redirect pointer
                *addrptr ← newbase + offset
                needstrace ← False                  ▷ Has been traced already
    else if addr is not null then
                        ▷ if addr is not in the heap, it must refer to a root
        if toplevel then
            newbase ← addr                          ▷ Top level refs point to the start
            type ← TypeOf(addr)
            needstrace ← True
        else
            needstrace ← False                      ▷ A top level call will trace it
    if needstrace then
        Trace(newbase, type)
```

Algorithm 4 Trace an item and preserve all items it keeps alive

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* $\in$ ADDRSINSIDE(*type*) **do**
      $aiaddr \leftarrow base + aioff$
      PRESERVEANDTRACE(*aiaddr*, `False`)

Algorithm 5 Trace a slice header, and preserve and trace its slice

Require: *shp*: Address of the slice header

```
function TraceSlice(shp)
    if Capacity(shp) = 0 then
        return
    slicestart ← Data(shp)
    if slicestart is in the heap then
        ⟨base, zhp⟩ ← LookupHeapArray(slicestart)
        type ← ElementType(TypeIn(zhp))
        es ← SizeOf(type)                                  ▷ Element size
        cap ← Capacity(shp)
        fse ← (slicestart − &Elements(base, 0))/es
        lse ← fse + cap − 1
        fce ← fse
        while 0 ≤ fce − 1 ∧ SeenPrev(base, fce − 1)) do
            fce ← fce − 1
        lce ← lse
        while lce + 1 < cap ∧ SeenPrev(base, lce + 1)) do
            lce ← lce + 1
        copybase ←
            PreserveElements(slicestart, fce, lce, fse, lse, es)
        Data(shp) ← &Elements(copybase, fse − fce)
        count ← Count(shp)
        for i ← 0 to count − 1 do
            if SeenCurr(base, fse + i) = 0 then
                SeenCurr(base, fse + i) ← 1
                SeenPrev(copybase, fse − fce + i) ← 1
                eltbase ← &Elements(copybase, fse − fce + i)
                for all aioff ∈ AddrsInside(type) do
                    aiaddr ← eltbase + aioff
                    PreserveAndTrace(aiaddr, False)
        for i ← count to cap − 1 do
            SeenCurr(base, fse + i) ← 1
            SeenPrev(copybase, fse − fce + i) ← 1
```

Algorithm 6 Preserve slice elements

Require: *slicestart*: Starting address of slice data
Require: *fce*: Index of the first contiguous element
Require: *lce*: Index of the last contiguous element
Require: *fce*: Index of the first slice element
Require: *lce*: Index of the last slice element
Require: *es*: The size of each element

function PRESERVEELEMENTS(*slicestart*, *fce*, *lce*, *fse*, *lse*, *es*)
  $\langle base, zhp \rangle \leftarrow$ LOOKUPHEAPARRAY(*slicestart*)
  if $\forall i \in fce..lce$. $\neg$SEENCURR(*base*, *i*) **then**
    $numelts \leftarrow lce - fce + 1$
    $copybytes \leftarrow \left\lceil \frac{headerbytes + \lceil (2 * numelts)/8 \rceil}{alignment} \right\rceil * alignment + numelts * es$
    $copybase \leftarrow$ ALLOCFROM(TOSPACE(*zhp*), *copybytes*)
    ARRAYNUMBYTES(*copybase*) $\leftarrow$ *copybytes*
    ARRAYNUMELTS(*copybase*) $\leftarrow$ *numelts*
    for $i \in 0..numelts - 1$ **do**
      SEENPREV(*copybase*, *i*) $\leftarrow fse \leq fce + i < lse$
      SEENCURR(*copybase*, *i*) $\leftarrow 0$
    COPYMEMORY(&ELEMENTS(*copybase*, 0), *slicestart*, $numelts * es$)
    $*$(&ELEMENTS(*base*, 0) + $fce * sz$) $\leftarrow$ *copybase*
  else
    copybase $\leftarrow *$(&ELEMENTS(*base*, 0) + $fce * sz$)
  return *copybase*

Chapter 6

Supporting RBMM in a Concurrent Environment

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

Neil deGrasse Tyson

Multicore 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 compute power. One such solution is to merely provide more CPU execution cores for processing (multicore systems). However, this leads to an interesting problem. The problem being: If a program is to achieve the best

performance, then programmers must write programs that take advantage of the increase in CPU parallelism.

Given the current status of computer hardware, it is tricky now to purchase a desktop or laptop that does not have multi-core support. Therefore, it is of interest for the programmer to take advantage of such an environment to attain the best performance out of the hardware presented.

Up to this point in this thesis, our work has covered introducing RBMM into a non-concurrent sequential subset of the Go programming language. In this chapter we introduce a design that allows RBMM to work within a coroutine environment as provided by the Go programming language. To our knowledge this is a unique contribution that introduces RBMM into a langauge whose concurrency is designed around the *Communicating Sequential Processes* formal language [36].

6.1 Introduction

Often the terms *parallel* and *concurrent* are confused. To clarify, a concurrent system is one that can execute multiple threads all at once. Process executions are interleaved (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. Similarly, 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 threading a program and allowing it to execute in a parallel context as *concurrent* or *parallel* programming. The slight differences in meaning are not necessary to explain our RBMM

modifications.

Both parallel and concurrent programming introduce additional complexity for the programmer to manage. Programmers must be aware of the challenges (deadlock and race-conditions) and to avoid them as best as possible.

While hardware seems to be increasing in performance, easy solutions to concurrent programming are still primitive. A programmer must prevent one thread from reading or writing data that another thread might be manipulating at the same exact time, if not then one of the threads (or possibly both) will have an incorrect picture of the program's state. This can lead to deadlock and race conditions. We call such invalid data access (one thread reading or writing to data that another thread might also be reading or writing) a *data contengency*.

As mentioned, no simple solutions have become common to avoid such situations. Mostly, the programmer must employ complicated concepts of locking and mutex 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! Not only must a programmer solve a problem but they might also have to manage resources (*e.g.* memory management), all while trying to accomplish these tasks utilizing concurrent contexts.

6.2 Go's Solution

The Go programming language aims to reduce concurrent programming complexity by having a semantics of sharing memory by communication [21]. To do this, Go relies on co-routines and channels as introduced by Tony Hoare's 1978 paper, *Communicating Sequential Processes* [36]. Such a system eliminates the need for a programmer to explicitly lock data shared amongst threads. Instead, a program can be written using channels and co-routines (go-routines) that permits atomic access to data.

6.3 Design

While Go programmers might have an easy means of applying concurrent models to their software, our implementing of parallel-safe RBMM must still rely on more traditional forms of handling concurrent data access. In fact, our RBMM-aware Go runtime is all written in C, therefore we can rely on complicated yet working constructs to prevent such 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 be made parallel by simply prefixing the 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.[1] 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 the transformation pass, when it sees the marks, should 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 our semantics originally described in Section 4.4.3. Both *send* and *recv* require channel variables. Since channels are allocated with `new`, they have regions. Go's "go" statements take a single function 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 those that return values. Go will not permit "go" statements to return actual values, thus

[1]The Go language and runtime 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. Early versions of the Go runtime provided by gcc did not support this multiplexing of go-routines, and instead placed each one onto its own system thread. For the purposes of this section, such details are not necessary.

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

Figure 6.1: Added semantics for handling Go's concurrency primitives.

syntactically enforcing the programmer to make use of channels. Later in this chapter we explain our requirement for having messages being sent and received to share the same region as the channel.

6.3.1 Gory Details of 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 respobnsible for creating a new thread and then calling the function that is to be made concurrent. This helper function takes a function pointer and a set of arguments. The function pointer is that of the 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 function, executes the function, and the warpper exits. Once the wrapper has executed, the go-routine can be thought of as being terminated.

6.3.2 Concurrency for Region Operations

As with our semantics, we must also take a few considerations as to the meanings of our region operations which we originally described in Section 4.4.2. Such considerations are to prevent any data contengencies when accessing the data that these operations manipulate (region metadata). Such accessess must be atomic, if not then the potential of a data contengenciy can arise.

Concurrency for IncrProtection() and DecrProtection()

Our concurrent RBMM system must take caution during region removal. We must prevent a thread from prematurely removing a region that might be shared amongst other regions. Such a case can occur if one go-routine has completed its use of a region, and then decrementing the protection counter and removing that region. Potentially another go-routine might still be using that region, and if it suddenly becomes invalid, then the resulting program is incorrect (and might terminate prematurely). To prevent such cases our protection counter takes on a slightly different meaning in the context of go-routines. Consider the following example:

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

    // More code...

    go processSomeData(val)

    // More code...
}
```

In the preceeding example, following our previous semantics of our protection counters, we would have incremented the counter just prior to calling

`processSomeData` and decremented it just after that call. Go-routine calls spawn off a new lightweight thread of execution and the current thread will procede to the next statement despite how long the go-routine'd function takes to execute. If we were following our old semantics we would have incremented the counter, called `processSomeData`, and immediatley decremented the counter. Possibly we might have removed the region associated to `val`, despite the knowledge that the concurrently executing go-routine with `processSomeData` might still need it. This could result in premature program termination.

To prevent such a case, we increment the protection counter just prior to making the go-routine call, as we would have done in our previous semantics. The adjustment we add here is that we decrement the counter once the go-routine completes. To understand our reasoning behind this adjustment it is necessary to understand how a go-routine operates at the low-level, as presented in Section6.3.1. We inset the decrement and region removal in the wrapper that is responsible for executing the call at runtime.

Concurrency for CreateRegion()

For region creation operations, our parallel modification allocates space for, and initializes, one additional fields in the region header: a lock variable that our runtime uses to prevent other threads from removing or concurrently allocating from the region when an allocation is taking place.

Concurrency for AllocFromRegion()

For 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.

Concurrency for RemoveRegion()

For region remove operations, our modification operates, under mutual exclusion, on the field in the region header that records the number of threads that contain references to the region. 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. 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.

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. 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 functions 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 that makes the last reference to a region in that thread, the question of *which* of these per-thread last references is actually executed last at runtime may depend not just on the input to the program but also on the effects of thread scheduling, and thus in general it cannot be decided statically.

The overall transformation is shown below 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 do to CPU 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 `main`, when `f'` exits, its thread will not have any remaining references to the regions it

$$\boxed{\texttt{go}\ \ f(v_1, \ldots v_n)\langle r_1, \ldots r_p\rangle;} \ \rightsquigarrow \ \boxed{\begin{array}{l}\mathsf{IncrProtection}(r_1); \ldots \\ \mathsf{IncrProtection}(r_p); \\ \texttt{go}\ \ f'(v_1, \ldots v_n)\langle r_1, \ldots r_p\rangle;\end{array}}$$

$$\begin{array}{ll}\text{where} & \texttt{func}\ f'(f'_1, \ldots f'_n)\langle r_1, \ldots r_p\rangle\ \{ \\ & \quad f(f'_1, \ldots f'_n)\langle r_1, \ldots r_p\rangle; \\ & \quad \mathsf{DecrProtection}(r_1); \ldots \mathsf{DecrProtection}(r_p); \\ & \quad \mathsf{RemoveRegion}(r_1); \ldots \mathsf{RemoveRegion}(r_p); \\ & \}\end{array}$$

Figure 6.2: Go-routine transformation

handles, but unlike `main`, it gets some regions from its parent thread, and does not have to create them all by itself.

Note that the 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.

Another optimization applies when a go-routine call site is the last reference to a region in the parent thread. In that case, the increment of the thread reference count at the call site and its decrement in the remove region

operation in the parent immediately afterward would cancel each other out, and thus both can be optimized away.

When a thread `t1` sends a message to another thread `t2`, with a statement such as `send` v_1`on` v_2, the code executed by `t1` effectively decides what region supplies the memory for the message: it will be $\mathsf{R}(v_1)$. When `t2` receives the message, it will do so with a statement such as v_3 = `recv from` v_4. After this statement, $t2$ will believe the message to be in region $\mathsf{R}(v_3)$. We need this to be the same as $\mathsf{R}(v1)$, since otherwise the two threads will disagree on when the region of the message can be reclaimed. We ensure this by imposing this chain of equalities: $\mathsf{R}(v_1) = \mathsf{R}(v_2) = \mathsf{R}(v_4) = \mathsf{R}(v_3)$. 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 id of the channel in a message. Our current setup stores all parts of a data structure in the same region, and this certainly applies to data structures sent as messages. 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 that is sent in a message on channel c_1, then $\mathsf{R}(c_1) = \mathsf{R}(c_2)$. 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.

6.3.3 Producer and Consumer Example

The following example illustrates the aforemention design presented in this chapter. We choose to use a common concurrecny pattern, a produce and consumer model. In this example a client *proudces* data at a pseudo-random interval of $[0, 3)$ seconds. A server listens, in a separate go-routine, and *consumes* any data it receives. Both the clients and server communicate via a shared channel. The main thread of execution creates the shared channel and passes it to all of the client and server go-routines that it also instantiates.

Figure 6.3 is the unmodified version without RBMM transformations.

6.4 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 earlier. We look at both time and space metrics to gauge the performance of our modifications.

6.4.1 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.

```
package main
import ("time"; "math/rand")

type Data struct {client_id, seq_id int; data [128]byte;}

func client(client_id int, ch chan *Data) {
    counter := 0
    for {
        counter++
        d := new(Data)
        d.client_id = client_id
        d.seq_id = counter

        ch <- d
        println("Client", client_id, "sent sequence", counter)
        time.Sleep(time.Duration(rand.Intn(3)) * time.Second)
    }
}

func server(ch chan *Data, done chan int) {
    sum := 0
    for {
        data := <-ch
        sum += data.seq_id
        if sum > 50 {
            done <- sum
        }
    }
}

func main() {
    ch := make(chan *Data)
    done := make(chan int)

    for i:=0; i<10; i++ {
        go client(i, ch)
    }

    go server(ch, done)
    <- done
}
```

Figure 6.3: Producer/consumer example

```
package main
import ("time"; "math/rand")

type Data struct {client_id, seq_id int; data [128]byte;}

func client(client_id int, ch chan *Data, reg *Region) {
    counter := 0
    for {
        counter++
        d := AllocFromRegion(reg, sizeof(Data))
        d.client_id = client_id
        d.seq_id = counter

        ch <- d
        println("Client", client_id, "sent sequence", counter)
        time.Sleep(time.Duration(rand.Intn(3)) * time.Second)
    }

    DecrThreadCnt(reg)
    RemoveRegion(reg)
}

func server(ch chan *Data, done chan int, reg1 *Region, reg2 *Region) {
    sum := 0
    for {
        data := <-ch
        sum += data.seq_id
        if sum > 50 {
            done <- sum
        }
    }

    DecrThreadCnt(reg1)
    DecrThreadCnt(reg2)
    RemoveRegion(reg1)
    RemoveRegion(reg2)
}

func main() {
    reg1 := CreateRegion()
    ch := AllocFromRegion(reg1, sizeof(chan *Data))

    reg2 := CreateRegion()
    done := AllocFromRegion(reg2, sizeof(chan int))

    for i:=0; i<10; i++ {
        IncrThreadCnt(reg1)
        go client(i, ch, reg1)
    }

    IncrThreadCnt(reg1)
    IncrThreadCnt(reg2)
    go server(ch, done, reg1, reg2)
    <- done
}
```

Figure 6.4: Producer/consumer example with region annotations

We added a few new tests which we have not looked at before: `k-nucleotide`, `regex-dna`, and `threadring`. 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. `threadring` 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 *(rbmm)* and compared that data to the output of the unmodified versions *(plain)* to ensure our results match that of the unmodified (*plain*) 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 (*HWM*) and *Elapsed Time* value. We also disabled our garbage collector, as we are less confident in its implementation. Instead, we relied on Go's existing GC as we did in Chapter 4.

The region count values are only available for the modified tests and these numbers were gathered once and not sampled 10 times. `threadring` intentionally terminates early by calling *os.Exit()*. Our statistical counters are only displayed upon return from *main*. For the `threadring` test, we had to run the application in the GNU debugger (GDB) and break on the *os.Exit* call. We then could look at our statistical counter's 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 goroutines, we cannot rely on this delta. For instance, a program can terminate before all goroutines complete. Often, the parent goroutine and its child goroutine(s)

are naturally synchronized via channels. The thunk that wraps the function call declared in the go statement will try to remove regions after the child has responded to its parent via channel communication. If the parents execution is quick enough, the program might terminate before the child goroutine and thunk 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 measure 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 an RBMM system might impose during program execution.

Our experiments were conducted on the same hardware as our previous two evaluations (see 4. That system is now running Ubuntu 13.04. The GCC used to compile both our modifications (our plug-in and runtime) 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 that 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.

6.4.2 Results

Table 6.1 shows our performance results from the aforementioned 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. We do appear to be comparable for programs with short execution times.

Our memory performance is comparable as well. In most cases we did not benefit the memory footprint of the process, usually to the order of a 10-20KB

Test	HWM(KB)	Elapsed(seconds)	Regions	Executable Size(KB)
fannkuch-parallel (plain)	8,764.80	4.58	NA	84
fannkuch-parallel (rbmm)	8,780.00	4.06	6	167
k-nucleotide-parallel (plain)	39,738.40	1.29	NA	87
k-nucleotide-parallel (rbmm)	44,086.00	1.23	15	170
regex-dna-parallel (plain)	26,845.60	2.46	NA	76
regex-dna-parallel (rbmm)	26,642.80	2.44	11	156
spectral-norm-parallel (plain)	8,957.20	0.81	NA	73
spectral-norm-parallel (rbmm)	8,975.60	0.86	181	159
threadring (plain)	14,016.80	6.17	NA	65
threadring (rbmm)	16,069.60	6.16	506	146

Table 6.1: Go-routine performance with and without RBMM

increase, as witnessed by `fannkuch-parallel` and `spectral-norm-parallel`. We also negatively impacted memory to the order of 4.3 MB for `k-nucleotide` and 2.1 MB for `threadring`. An increase in memory is not unheard of for an RBMM system, and is the result of the *region-bloat* problem where the region keeps unused data around until to the program point where our static analysis found a place to safely reclaim the region. We note that adding our goroutine-capable RBMM system adds on average 86.6 KB to the binary file size of the benchmark.

6.5 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 a thread counter which represents 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 any imperatively styled language, especially those with a similar CSP-styled concurrency system.

Chapter 7

Literature Review

From the front to the back as pages turn; reading is a very fresh way to learn.

Run-D.M.C

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 [37]. 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 looks for unreachable objects when the system runs out of free memory. Their

collector performs its task by scanning memory, starting at the base system registers, looking for any objects which can no longer be accessed. These registers form the root-set for their collector. All objects that can be reached have their sign-bit set (the marking phase). The collector can then begin its sweeping pass by scanning the entire memory space and returning any unused (unmarked) memory back onto a freelist for later allocations to use. 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" [37].

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 in memory also happen to be the objects that are the most recently allocated [63]. One method to reduce the overhead of GC is to scan a subset of the entire memory space for collection. Generational garbage collectors exploit this observation. Ungar's generational scanner, for Berkley Smalltalk, shows a notable improvement over past GC techniques, such as reference counting (33% improvement over deferred reference counting). Reference counting is a technique used by some GC algorithms to determine when an object is no longer reachable. Each time an object is referred to during execution (*e.g.*, assignment) its counter is incremented, and 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. 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 objects survive a number of collection cycles they are promoted to an older generation, which is scanned less frequently. This copying compacts the memory space and enhances data locality. Ungar observed that the copying of surviving objects to a newer generation has lower overhead than scanning the dead-/unreachable objects.

Lieberman and Carl introduced a real-time GC algorithm for partitioning the memory space into regions based on object lifetimes [45]. Regions belong to a specific generation 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 be reclaimed. Their algorithm makes use of the fact that in certain systems (*e.g.*, Lisp) pointers more commonly 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. Barth investigated the use of static analysis to benefit GC [4]. 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.

In 1988, Ruggieri and Murtagh introduced lifetime analysis [51]. This static analysis technique can obtain an upper-bound on how long any particular object lives. With this knowledge, objects having lifetimes of decidable bounds can be allocated at function entry and deallocated at function exit, reducing the need (and overhead) for GC.

Barry Hayes showed that the weak generational hypothesis holds; young objects have short lifetime [30]. However, he found that waiting longer to perform reclamation is not useful. His data suggests that age based lifetime is not an adequate measure when considering 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 a cluster as being a "key." When that key object is 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 [33]. This information can be used to insert deallocation calls into the program at compile-time, which he found could fulfill 80-100% of object allocations. This result is a measure of certain classes of programs and is not representative of all programs. This finding strengthens the motivation for combining static analysis with a runtime garbage collector.

Aside from lifetime, object connectivity is another property of dynamically allocated data that can be used to optimize GC. In 2002 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 [35]. The latter is also a property of object lifetime. 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 [34].

Khedker et al. demonstrated that a static analysis can be used to benefit time and space properties of GC [40]. 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 may allow the garbage collector to collect the object earlier. The authors found the latter to reduce heap size, and they postulate that this 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 the latter is true by using a threshold algorithm. These thresholds act as a dynamically changing limit of the heap size [2]. 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. Their 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, 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 a collection cycle [39].

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 [27]. The collector is based on Baker's copying collector [3], which divides the memory space into semispaces (sometimes called *fromspace* and *tospace*) [16].

The Baker algorithm works by dividing the memory into two semispaces and then coloring the objects in those spaces: *white* objects are those in the *fromspace*, *black* objects are those that have been traced and copied, and *grey* objects are those that have been copied to *tospace* but have not been traced [3]. Baker mentions that, even though his copying collector compacts freespace and helps to reduce memory fragmentation, it can require excessive memory for some programs. The latter are natural properties of semispace copying collectors; they require at least enough memory to support worse case copying. This being the case when all allocated objects survive a GC cycle and must be copied from *fromspace* to *tospace*.

In 1993, Doligez and Leroy implemented a parallel GC for Concurrent Caml Light (a derivative of ML) [14]. 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 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.

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 more efficiently manage memory. 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 [5]. They also presented a new system, called a reap allocator, which acts as a combination of both general purpose and region allocators. Reaps begin 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, then the region continues allocating from its end. Their results 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 may not be able to replicate the performance of these manually tuned region allocators.

Over a course of nearly nine years, a vast amount of RBMM research was conducted by the efforts of Tofte, Talpin, Hallenberg, Birkedal, and Elsman with their extensive exploration for improving the memory system of the Standard ML programming language [60]. The initial motivation for creating

a RBMM system was to reduce memory footprints 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 language semantics and static analysis for inferring sets of referenced data (regions) in an extension of Core ML. They mention that such a system can be used for managing memory [58]. Soon after this research was published, Tofte and Talpin introduced RBMM for Standard ML [61]. Their seminal idea was to allocate data, based on lifetimes, into *stacks* of regions. They found that the maximum resident memory size often favored the RBMM approach, while GC was often faster. The authors later proved the soundness of their semantics [62]. 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 [59].

While RBMM might have been said to be birthed by the Tofte and Talpin team, this was not the first use of regions. Rather, this was more of a birth of the *region-based memory management* term. In 1990 David Hanson published an article in Software Practices and Experience about his *arena* memory management system [29]. 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 is nearly zero sum, 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 a stack-based RBMM system [1]. 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 [32]. 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[46]. 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 suggests that this was the result of RBMM performing 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 [26]. Unlike our approach that we present later in this thesis, enabling 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[15, 26]. 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 of 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 their implementation, by eliminating dangling pointers in their region typing system.

In 1998, Christiansen and Velschow investigated adding region semantics into a Java-like language they designed called "RegJava" [10]. 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, and 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 dependant 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. Salaganac, et al. implemented a "fast" static analysis to aid their goal of implementing RBMM on an embedded Java framework [52]. This static analysis can be used to implement a semi-automated region inference algorithm [53]. Their system detects memory leaks which then produces feedback to the developer, in order to prevent what the authors term, "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 [59]. 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). The authors 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. The latter exposes a limitation of their analysis.

Another Java implementation using RBMM is that of Cherem and Rugina [7]. Like Salaganac et al., Cherem and Rugina's approach is based on a points-to analysis and also suffers from the memory leak problem mentioned earlier [53]. 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 also investigated implementing RBMM in a real-time Java framework capable of handling multithreaded applications. Their approach utilizes combined regions and ownership types [6]. 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 [19], forms a hierarchy of regions and is safe from dangling pointers. This system also allows multiple processes to share region data.

Chin et al. also implemented a system of region inference for Java which is based on the Real-Time Java specification [9]. 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 region annotated 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 [28]. 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. The authors 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 small lifetime objects inside longer lived scopes. This is the same as the region bloat problem discussed in Chapter 3.

Stoutamire researched a similar RBMM concept for the Sather programming language to improve memory locality [56]. 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. Results from a partial implementation of the author's 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.

Gay and Aiken 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 [18]. 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@) [18].

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 [25]. Cyclone does provide automatic default annotations. Their 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.*, http client) applications; however, the overhead was significant (from 2.07-2.85 times slower) for some compute intensive benchmarks.

Lattner and Adve 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", than using malloc [42, 43].

The logic programming language Mercury was implemented with an automated RBMM system, in contrast to its existing garbage collector [49]. Their stackless design was inspired by Cherem and Rugina's Java RBMM implementation, to permit shorter-lived regions [48]. Their benchmarks ran 25% faster using RBMM than with the Boehm GC [50]

Parallel processes and threads can complicate how data is accessed, and the same goes 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 another process might access the same piece of data that another process might also be accessing. The term "access" refers to memory

reads and writes. Gay and Aiken mention that implementing RBMM in a parallel system should be trivial, as the only locking (to prevent concurrent data mutation) need to occur on the region creation and removal operations [17]. In fact, Seidel and Vojdani 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 [54]. 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 [19, 20]. If a processes has a lock on a memory resource, or region, no other competing processes 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 improvements. The key point is that they implemented a safe parallel system using RBMM. Adve and Lattner 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 make them parallel [42].

Chapter 8

Conclusion

This thesis has explored the capabilities that an alternative form for automatic memory management can provide 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 additional programmer code annotation aside from what the language already provides (*new* and *make*). Thus, we maintain the simplicity for the programmer that Go provides; The programmer is removed 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 4 we introduced RBMM concepts and our design via a simple syntax and semantics. Our goal was to use our RBMM system in Go to reduce time and space overhead that Go's existing GC environment would produce. We have shown that our performance numbers using RBMM are comparable to unmodified Go benchmarks running with their garbage collector. For one case in particular, `binary-tree`, we can significantly reduce execution time by reducing time spent in GC by using regions. We also show that 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. MD:▶**Please make sure that I am not lying here. The**

last two points about efficiency and small regions are based on the binary-tree results from our first paper◀ .

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 [11]. 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. In other words, acting 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 reclaim of sequences of items within Go's slice primitive. While we spent much time implementing a region-aware garbage collector, our result was buggy and less than effective. Our system does have a bug which we have spent much time on trying to correct, but to little avail. Our results show that our bug is not directly garbage collector related.

Our concepts and designs can be extracted from this research and made production quality. These ideas can be used,not only for Go, but other languages. Since all of our work transforms GCC's intermediate GIMPLE and RTL languages, it should be relatively easy to modify our code to support other programming languages that GCC can input.

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 shortly lived benchmarks we evaluated.

Bibliography

[1] Alex Aiken, Manuel Fähndrich, and Raph Levien. Better static memory management: Improving region-based analysis of higher-order languages. In *Proc. PLDI 1995*, pages 174–185. ACM Press, 1995.

[2] Eshrat Arjom and Chang Li. Controlling garbage collection and heap growth to reduce execution time of java applications. In *In ACM Conference on ObjectOriented Programming, Systems, Languages, and Applications (OOPSLA01)*, 2001.

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

[4] Jeffrey M. Barth. Shifting garbage collection overhead to compile time. *Commun. ACM*, 20(7):513–518, July 1977.

[5] Emery D. Berger, Benjamin G. Zorn, and Kathryn S. McKinley. Reconsidering custom memory allocation. In *OOPSLA 2002*, pages 1–12. ACM Press, 2002.

[6] Chandrasekhar Boyapati, Alexandru Salcianu, William Beebee Jr., and Martin Rinard. Ownership types for safe region-based memory management in real-time Java. In *Proc. PLDI 2003*, pages 324–337. ACM Press, 2003.

[7] Sigmund Cherem and Radu Rugina. Region analysis and transformation for Java programs. In *Proc. 4th ISMM*, pages 85–96. ACM Press, 2004.

[8] Dmitry Chestnykh. Passwordhash and PBKDF2 Go libraries.

[9] Wei-Ngan Chin, Florin Craciun, Shengchao Qin, and Martin Rinard. Region inference for an object-oriented language. *SIGPLAN*, 39(6):243–254, 2004.

[10] Morten V. Christiansen and Per Velschow. Region-based memory management in Java. Technical report, 1998.

[11] Matthew Davis, Peter Schachte, Zoltan Somogyi, and Harald Søndergaard. A low overhead method for recovering unused memory inside regions. In *Proceedings of the 2013 ACM SIGPLAN Workshop on Memory Systems Performance and Correctness*, MSPC '13, New York, NY, USA, 2013. ACM.

[12] Michal Derkacz. BLAS: Basic linear algebra subprograms for Go.

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

[14] Damien Doligez and Xavier Leroy. A concurrent, generational garbage collector for a multithreaded implementation of ml. In *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '93, pages 113–123, New York, NY, USA, 1993. ACM.

[15] Martin Elsman. Garbage collection safety for region-based memory management. In *Proceedings of the 2003 ACM SIGPLAN International Workshop on Types in Language Design and Implementation*, pages 123–134. ACM, 2003.

[16] Robert R. Fenichel and Jerome C. Yochelson. A lisp garbage-collector for virtual-memory computer systems. *Commun. ACM*, 12(11):611–612, November 1969.

[17] David Gay and Alex Aiken. Memory management with explicit regions. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, pages 313–323, New York, NY, USA, 1998. ACM Press.

[18] David Gay and Alex Aiken. Language support for regions. In *Proc. PLDI 2001*, pages 70–80. ACM, 2001.

[19] Prodromos Gerakios, Nikolaos Papaspyrou, and Konstantinos Sagonas. A concurrent language with a uniform treatment of regions and locks. In *Electronic Procedings in Theoretical Computer Science*, pages 79–93, 2010.

[20] Prodromos Gerakios, Nikolaos Papaspyrou, and Konstantinos Sagonas. Race-free and memory-safe multithreading: Design and implementation in Cyclone. In *Types in Languages Design and Implementation*, pages 15–26, 2010.

[21] Andrew Gerrand. Share memory by communicating, 2010.

[22] Google. The go programming language specification, September 2012.

[23] Google. Effective go, February 2013.

[24] Google. The go programming language faq 1.0.3, February 2013.

[25] Dan Grossman, Greg Morrisett, Trevor Jim, Michael Hicks, Yanling Wang, and James Cheney. Region-based memory management in Cyclone. In *Proc. PLDI 2002*, pages 282–293. ACM Press, 2002.

[26] Niels Hallenberg, Martin Elsman, and Mads Tofte. Combining region inference and garbage collection. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, pages 141–152, New York, NY, USA, 2002. ACM.

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

[28] H. Hamza and S. Counsell. Rtsj scoped memory management: State of the art. *Science of Computer Programming*, 77(5):644–659, 2012.

[29] D. R. Hanson. Fast allocation and deallocation of memory based on object lifetimes. *Software Practice and Experience*, 20(1):5–12, January 1990.

[30] Barry Hayes. Using key object opportunism to collect old objects. In *Conference proceedings on Object-oriented programming systems, languages, and applications*, OOPSLA '91, pages 33–46, New York, NY, USA, 1991. ACM.

[31] Barry Hayes. Finalization in the collector interface. In *Memory Management*, pages 277–298. Springer, 1992.

[32] Fritz Henglein, Henning Makholm, and Henning Niss. A direct approach to control-flow sensitive region-based memory management. In *Proceedings of the 3rd International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming*, pages 175–186. ACM, 2001.

[33] James Edward Hicks Jr. *Compiler-Directed Storage Reclamation Using Object Lifetime Analysis*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, 1992.

[34] Martin Hirzel, Amer Diwan, and Matthew Hertz. Connectivity-based garbage collection. In *Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, pages 359–373. ACM Press, 2003.

[35] Martin Hirzel, Johannes Henkel, Amer Diwan, and Michael Hind. Understanding the connectivity of heap objects. In *ACM SIGPLAN Notices*, volume 38, pages 36–49. ACM, 2002.

[36] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, August 1978.

[37] et al. John McCarthy. Artificial intelligence. Technical report, Research Laboratory of Electronics, Massachusetts Institute of Technology, April 1959.

[38] Richard Jones. The memory management glossary, 2001.

[39] Richard Jones, Anthony Hosking, and Eliot Moss. *The Garbage Collection Handbook: The Art of Automatic Memory Management*. CRC Press, 2012.

[40] Uday P. Khedker, Amitabha Sanyal, and Amey Karkare. Heap reference analysis using access graphs. *ACM Transactions on Programming Languages and Systems*, 30(1):Article 1, 2007.

[41] Leslie Lamport. Garbage collection with multiple processes: an exercise in parallelism. In *Proc. of the 1976 International Conference on Parallel Processing*, pages 50–54, 1976.

[42] Chris Lattner and Vikram Adve. Automatic pool allocation for disjoint data structures. *SIGPLAN Notices*, 38:13–24, 2003.

[43] Chris Lattner and Vikram Adve. Automatic pool allocation: Improving performance by controlling data structure layout in the heap. In *Proc. PLDI 2005*, pages 129–142. ACM Press, 2005.

[44] Heng Li. Programming language benchmarks.

[45] Henry Lieberman and Carl Hewitt. A real-time garbage collector based on the lifetimes of objects. *Commun. ACM*, 26(6):419–429, June 1983.

[46] Henning Makholm. A region-based memory manager for prolog. In *Proceedings of the 2nd international symposium on Memory management*, ISMM '00, pages 25–34, New York, NY, USA, 2000. ACM.

[47] Andre Moraes. Gocask library.

[48] Quan Phan. *Region-Based Memory Management for the Logic Programming Language Mercury.* PhD thesis, Catholic University of Leuven, Belgium, 2009.

[49] Quan Phan and Gerda Janssens. Towards region-based memory management for Mercury programs. In *In CICLOPS*, 2006.

[50] Quan Phan, Zoltan Somogyi, and Gerda Janssens. Runtime support for region-based memory management in mercury. In *Proceedings of the 7th international symposium on Memory management*, pages 61–70. ACM, 2008.

[51] C. Ruggieri and T. P. Murtagh. Lifetime analysis of dynamically allocated objects. In *POPL '88: Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 285–293, New York, NY, USA, 1988. ACM.

[52] G. Salagnac, S. Yovine, and D. Garbervetsky. Fast escape analysis for region-based memory management. *Electronic Notes on Theoretical Computer Science*, 131:99–110, May 2005.

[53] Guillaume Salagnac, Christophe Rippert, and Sergio Yovine. Semi-automatic region-based memory management for real-time Java embedded systems. In *Proceedings of the 13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 73–80. IEEE Computer Society, 2007.

[54] Helmut Seidl and Vesal Vojdani. Region analysis for race detection. In Jens Palsberg and Zhendong Su, editors, *Static Analysis*, volume 5673 of *Lecture Notes in Computer Science*, pages 171–187. Springer Berlin Heidelberg, 2009.

[55] G.L.J. Steele. *Data Representations in PDP-10 MacLISP.* AI Memo. Artificial Intelligence Laboratory, MIT, 1977.

[56] David Stoutamire. *Portable, Modular Expression of Locality*. PhD thesis, University of California, Berkely, 1997.

[57] Mark Summerfield. *Programming in Go: Creating Applications for the 21st Century*. Addison-Wesley Professional, 2012.

[58] Jean-Pierre Talpin and Pierre Jouvelot. Polymorphic type, region and effect inference. *Journal of Functional Programming*, 2:245–271, 1992.

[59] Mads Tofte and Lars Birkedal. A region inference algorithm. *ACM Transactions On Programming Languages and Systems*, 20, 1998.

[60] Mads Tofte, Lars Birkedal, Martin Elsman, and Niels Hallenberg. A retrospective on region-based memory management. *Higher-Order and Symbolic Computation*, 17(3):245–265, 2004.

[61] Mads Tofte and Jean-Pierre Talpin. Implementation of the typed call-by-value lambda-calculus using a stack of regions. In *Proc. 21st POPL*, pages 188–201. ACM Press, 1994.

[62] Mads Tofte and Jean-Pierre Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, 1997.

[63] David Ungar. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. In *Proceedings of the first ACM SIGSOFT/SIGPLAN software engineering symposium on Practical software development environments*, SDE 1, pages 157–167, New York, NY, USA, 1984. ACM.

[64] Todd R. Weiss. Out-of-memory problem causes mars rover's glitch. February 2004.

[65] Peter Zijlstra. High memory handling. *Linux Kernel 3.8.4 Source Code Documentation*, 2013.