Towards Region-Based Memory Management for Go

Matthew Davis, Peter Schachte, Zoltan Somogyi, and Harald Søndergaard

Department of Computing and Information Systems and NICTA Victoria Laboratories
The University of Melbourne, Victoria 3010, Australia
mattdavis9@gmail.com, {schachte,zs,harald}@unimelb.edu.au

Abstract

Region-based memory management aims to lower the cost of deallocation through bulk processing: instead of recovering the memory of each object separately, it recovers the memory of a region containing many objects. It relies on static analysis to determine the set of memory regions needed by a program, the program points at which each region should be created and removed, and, for each memory allocation, the region that should supply the memory. The concurrent language Go has features that pose interesting challenges for this analysis. We present a novel design for region-based memory management for Go, combining static analysis, to guide region creation, and lightweight runtime bookkeeping, to help control reclamation. The 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 approach more practical than existing RBMM systems. Our prototype implementation covers most of the sequential fragment of Go, and preliminary results are encouraging.

Categories and Subject Descriptors D.3.2 [*Programming Languages*]: Language Classifications—Concurrent languages; D.3.4 [*Processors*]: Memory management (garbage collection); D.4.2 [*Operating Systems*]: Garbage collection

General Terms Languages, performance, experiments, design

Keywords Go, memory management, memory regions, program analysis, program transformation

1. Introduction

Memory management, the allocation and deallocation of the memory used to store the objects a program manipulates, is crucial to the correct and efficient operation of most programs. If a program fails to deallocate an object, or deallocates it long after it is last needed, the program will use too much memory, possibly reducing performance or even causing the program to run out of memory. If the program reclaims an object *before* it is last needed, the program may crash or produce incorrect results. Traditionally, in C-like languages, programmers manage memory manually, and have to figure out by themselves when an object should be freed. This is difficult, because whether or not a function can free a particular object depends not only on the behaviour of that function, but also on whether or not *any* code executed subsequently will need that object. This may be different for different calls to the function, and it may change as other parts of the program are modified.

Automatic memory management removes this burden from programmers. Long offered in functional and logic programming languages, its popularity has spread to imperative languages such as Java, C#, and Go. The usual form of automatic memory management is garbage collection (GC). A GC system periodically finds all allocated memory blocks that are reachable from registers, the stack, and global variables, and makes all other allocated blocks (the garbage) available for reuse. This is safe, as inaccessible objects will certainly not be used again, but it is also overly conservative, as some reachable blocks may not be used again. GC also uses significant computation time, so it is run infrequently. This allows allocated but unused memory to build up for a while before finally being reclaimed, increasing the amount of memory needed to carry out the computation, and reducing cache performance. Also, many garbage collectors do not work if the memory they work on is changed as they operate. This is fine for sequential programs, since execution waits while garbage is collected, but making all threads in a parallel program wait during GC significantly reduces the benefits of parallelism. So, languages that support multithreading tend to prefer GC algorithms that can operate while other cores mutate memory. While such GC algorithms do not need to "stop the world" during collections, they have higher overheads.

Region-based memory management (RBMM) is an alternative to GC that aims to reduce overheads and memory footprint. While GC works almost entirely at runtime, with relatively little compiler support (mostly in the form of enforcing rules such as not hiding pointers from GC), RBMM systems work mostly at compile time. RBMM systems analyse the program statically to determine which parts of the program's memory can be freed at the same time. They then transform the program, by inserting code to allocate those structures together, in the same *region* of memory, and to reclaim the whole region in a single operation, making all the memory in that region available for reuse.

An advantage of RBMM is that it does not require a scan of all of memory to determine which blocks can be reclaimed. Moreover, it can reclaim large chunks of memory in one fell swoop, rather than releasing small blocks one by one. It can also use less memory than GC, because it needs less for its own bookkeeping, and because it decides what memory can be reclaimed based on what the program may need to use in the future, rather than on what memory the program could possibly have access to. This means memory may be reclaimed more frequently, leading to more consistent memory usage. In contrast, a GC system will use more and more memory before periodically reclaiming some of it. It may also happen, however, that static analysis is unable to distinguish lifetimes well enough. This may put most of the memory allocated by the program into a single giant region, which cannot be released until very near the end of the computation.

MSPC'12 June 16, 2012, Beijing, China.

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

Figure 1. A representative Go/GIMPLE fragment

The original work on RBMM targeted Standard ML [16, 17]. The approach has since been applied to other languages, such as C, C++ [2, 10], Java [3, 4], and Mercury [15]. 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 so far with implementing RBMM for the programming language Go. A prominent feature of Go are "goroutines": independently scheduled threads. Goroutines may share memory, or they may communicate via named channels, *à la* Hoare's Communicating Sequential Processes (CSP). 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 is handed back to f's caller.

Go programmers are required to request dynamic memory via `new` or its variant `make`, which are like `malloc` in C. The Go implementation itself uses these primitives. 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. Our aim is to replace GC by RBMM as far as possible.

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 1. Reflecting the fact that we deal with three-address code, we have normalised the fragment in obvious ways, requiring for example that selectors, indexing, and binary operations are applied to variables, rather than to arbitrary expressions.

In this paper, we describe an approach to RBMM for a concurrent programming language. The approach is characterised by a new combination of static analysis to guide region creation and lightweight runtime bookkeeping to control reclamation. The novelty, and main advantage, of the 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 approach more practical than existing RBMM systems.

After an introduction to RBMM in Section 2, we show how we analyse Go programs to see what regions they need (Section 3). In Section 4, we show how to transform a program to utilise those regions. In Section 5 we present performance measurements of an early implementation of our approach. In Section 6, we briefly discuss related work, and Section 7 concludes.

2. 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 calls to the functions implementing the primitive operations on regions. Since one cannot allocate from a nonexistent region, and 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 block 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 overall overheads, we want to amortize the cost of the operations on a region over a significant number of memory blocks. Having each memory block stored in its own region would impose unacceptably high overheads, though it would also ensure that its storage is recovered as soon as possible. Having all memory blocks 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 blocks.

We now introduce some concepts that help explain the runtime support for regions. A *region page* 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.) A small part is a link field, so that pages can be chained into a linked list. A *region* is such a linked list. We reserve the initial part of the first page of a region's page list to hold the *region header*, which has bookkeeping information about the region, such as its most recent page and the next available word in that page. As we explain later, it also includes a mutex and some counts. 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*.

The run-time 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. Reclamation of a region simply means returning its list of pages to the freelist.

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

- CreateRegion(): Create an empty region from which data structures 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

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

Figure 2. Region constraint generation

until after $\mathsf{DecrProtection}(r)$ is called. We explain the role of this operation in Section 4.3.

- $\mathsf{DecrProtection}(r)$: Decrement the region's protection count.
- $\mathsf{IncrThreadCnt}(r)$: Increment the count of threads that contain references to r, ensuring that calls to $\mathsf{RemoveRegion}(r)$ do not actually reclaim r until after $\mathsf{DecrThreadCnt}(r)$ is called. We explain the role of this operation in Section 4.5.
- $\mathsf{DecrThreadCnt}(r)$: Decrement r's thread reference count.

3. Program Analysis

The job of our analyses is to decide, for each pointer-valued variable in the program, which region should hold the objects to which it points. We associate 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 associate a region variable even with non-pointer-valued variables. If v_2 is 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 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 equality constraints on these region variables; 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 invent 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 2 defines the functions we use to generate region constraints. The top of the figure gives the types of these functions. In these types, $\mathit{EqConstrs}$ is the set of equivalence constraints on region variables (each constraint is itself a conjunction of primitive equivalences); and $\mathit{Map} = \mathit{Fname} \to \mathit{EqConstrs}$ 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.

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.

Given the assignment $v_1 = v_2$ where v_1 and v_2 are pointers or structures containing pointers, they can obviously refer to the same memory, so we constrain them to get 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. Our system does not yet incorporate this refinement, though we are working on it. Instead, we simply require v_1 and v_2 to be 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. Since channels are allocated with `new`, they have regions. Later, in section 4.5, it will become clear why we require messages being sent and received to have the same region as the channel.

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, including the one representing the return value. 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 $\mathcal{P}$. Beginning 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. We repeat this computation, beginning each iteration with the ρ just computed, until the resulting ρ is the same in the previous iteration.

Figure 3 is an example program, for which analysis produces these constraints: for `CreateNode`: $\mathsf{R}(\texttt{CreateNode}_0) = \mathsf{R}(\texttt{n})$, for `BuildList`: $\mathsf{R}(\texttt{n}) = \mathsf{R}(\texttt{BuildList}_1) \wedge \mathsf{R}(\texttt{CreateNode}_0) = \mathsf{R}(\texttt{n})$ (some additional constraints will occur for temporary variables introduced in the GIMPLE code, but we ignore those here), and for `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, 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 reanalysing and recompiling any part of the program. With a context *insensitive* analysis, only modules that import a changed module will need to be reanalysed 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. 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 primitive with the

```
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 3. 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. Same program with region annotations

RBMM memory allocator, and inserting calls to create and remove regions. To support this, we must also transform functions to take regions as inputs next to the arguments they are written to expect.

As discussed in Section 3, our analysis only summarises the region equality 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 (so to minimise memory usage, it should be reclaimed). 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 again after the call. This small runtime overhead is the price we pay for limiting ourselves to a context insensitive program analysis. Figure 4 shows the automatically transformed version of the code in Figure 3.

To store objects with undetermined lifetimes, we define a single special region called the *global region*. This region exists for the duration of the computation. Data allocated in the global region can only be reclaimed by garbage collection, so it is actually allocated using Go's normal memory allocation primitives.

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. $\mathsf{compress}_f\langle r_1, \ldots r_n\rangle$ is the list of regions $\langle r_1, \ldots r_n\rangle$, except for the removal of duplicates, as implied by the region equality constraints for f's formal parameters and return value. $\mathsf{reg}(f)$ 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 $\mathsf{ir}(f) = \mathsf{compress}_f\langle \mathsf{R}(f_1), \ldots \mathsf{R}(f_n), \mathsf{R}(f_0)\rangle$. (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.

4.1 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.

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

4.2 Function Calls and Declarations

Every function that takes data structures as input or returns them as output must be transformed to also expect region arguments. We indicate the region arguments of a function by enclosing them in angle brackets following the ordinary function arguments:

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

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 that holds a structure, plus one if the result is a structure. 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:

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

where $\langle r_1, \ldots r_p\rangle = \mathsf{ir}(f)$

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:

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

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

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.

4.3 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 value, it may allow the region to remain after the function completes execution. This is handled by the following transformations.

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

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

$R = \{\mathsf{RemoveRegion}(r);\ |\ r \in \mathsf{reg}(f) \setminus \{\mathsf{R}(f_0)\}\}$

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.

$$\boxed{\begin{array}{l}r\text{=}\mathsf{CreateRegion}();\\ S_1; \ldots S_m;\\ S_{m+1}; \ldots S_n;\end{array}} \rightsquigarrow \boxed{\begin{array}{l}S_1; \ldots S_m;\\ r\text{=}\mathsf{CreateRegion}();\\ S_{m+1}; \ldots S_n;\end{array}}$$

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

$$\boxed{\begin{array}{l} S_1; \ldots S_m; \\ S_{m+1}; \ldots S_n; \\ \mathsf{RemoveRegion}(r); \end{array}} \rightsquigarrow \boxed{\begin{array}{l} S_1; \ldots S_m; \\ \mathsf{RemoveRegion}(r); \\ S_{m+1}; \ldots S_n; \end{array}}$$

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 could lead to out-of-memory errors, we push region creation and removal (as a pair) into loops where possible. We also push region creation and removal into conditionals where possible, because it can reduce peak memory use.

$$\boxed{\begin{array}{l} r = \mathsf{CreateRegion}(); \\ \texttt{loop}\ \{ \\ \quad S_1; \ldots S_m; \\ \} \\ \mathsf{RemoveRegion}(r); \end{array}} \rightsquigarrow \boxed{\begin{array}{l} \texttt{loop}\ \{ \\ \quad r = \mathsf{CreateRegion}(); \\ \quad S_1; \ldots S_m; \\ \quad \mathsf{RemoveRegion}(r); \\ \} \end{array}}$$

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

$$\boxed{\begin{array}{l} r = \mathsf{CreateRegion}(); \\ \texttt{if}\ t\ \{ \\ \quad S_1; \ldots S_m; \\ \}\ \texttt{else}\ \{ \\ \quad S_{m+1}; \ldots S_n; \\ \} \end{array}} \rightsquigarrow \boxed{\begin{array}{l} \texttt{if}\ t\ \{ \\ \quad r = \mathsf{CreateRegion}(); \\ \quad S_1; \ldots S_m; \\ \}\ \texttt{else}\ \{ \\ \quad r = \mathsf{CreateRegion}(); \\ \quad S_{m+1}; \ldots S_n; \\ \} \end{array}}$$

$$\boxed{\begin{array}{l} \texttt{if}\ t\ \{ \\ \quad S_1; \ldots S_m; \\ \}\ \texttt{else}\ \{ \\ \quad S_{m+1}; \ldots S_n; \\ \} \\ \mathsf{RemoveRegion}(r); \end{array}} \rightsquigarrow \boxed{\begin{array}{l} \texttt{if}\ t\ \{ \\ \quad S_1; \ldots S_m; \\ \quad \mathsf{RemoveRegion}(r); \\ \}\ \texttt{else}\ \{ \\ \quad S_{m+1}; \ldots S_n; \\ \quad \mathsf{RemoveRegion}(r); \\ \} \end{array}}$$

Our final region creation and removal transformation may be useful when only one arm of a conditional uses a region:

$$\boxed{\begin{array}{l} S_1; \ldots S_m; \\ r = \mathsf{CreateRegion}(); \\ \mathsf{RemoveRegion}(r); \\ S_{m+1}; \ldots S_n; \end{array}} \rightsquigarrow \boxed{\begin{array}{l} S_1; \ldots S_m; \\ S_{m+1}; \ldots S_n; \end{array}}$$

4.4 Region Protection Counting

To remove each region at the earliest possible time, we must put a call $\mathsf{RemoveRegion}(r)$ 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 reanalysed 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 (analysing callees before callers, and analysing mutually recursive functions together). This is simple 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 ultimately 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:

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

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.

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

We have not yet implemented this transformation. More importantly, we plan 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 object or region. For example, in RC [7], 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.

4.5 Goroutines

A Go program can create a new thread of execution by simply prefixing a function call (to a function that does not return a value) with the keyword `go`. The new function invocation will then execute in a new independently-scheduled thread, which will terminate when the call returns. 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. Our analysis marks regions passed in such calls, and our transformation, when it sees the marks, generates calls to modified versions of the region creation, allocation and removal operations.

For creation operations, the modification allocates spaces for and initializes two additional fields in the region header: a mutex, and count of threads referring to the region.

For allocation operations, the modification turns the usual code of the operation into a critical section that is protected by the mutex field in the region header, though this extra synchronization can be optimized away on allocation operations in the main thread before the first goroutine call involving the region.

For remove operations, the 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 the region is created, we initialize this field to one. When the region is mentioned as an argument in a goroutine call, we increment this counter. 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 the counter. If it is still positive, some other threads are still using the region, so the remove operation will actually reclaim the memory of the region only if the counter has gone to zero. This runtime test is necessary because, while a static analysis can figure out the 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 accidents of scheduling, and thus in general it cannot be decided statically.

The overall transformation is shown below. (The function invoked by a goroutine cannot return a value.) Our analysis treats the spawned-off function `f'` a bit like we treat `main`. Like `main`, when `f'` exits, its thread will not have any remaining references to the regions it handles, but unlike `main`, it gets some regions from its parent thread, and does not have to create them all itself.

Note that the increments must be done 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.

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

where
```
func f'(f'_1, ... f'_n)⟨r_1, ... r_p⟩ {
    dummyvar = f(f'_1, ... f'_n)⟨r_1, ... r_p⟩;
    DecrThreadCnt(r_1); ... DecrThreadCnt(r_p);
    RemoveRegion(r_1); ... RemoveRegion(r_p);
    return dummyvar;
}
```

We can optimize the above code in some cases. For example, in some cases we can guarantee that some per-thread last references cannot be the last reference globally. For example, if two threads communicate using a 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 DecrThreadCnt in the reading thread. In fact, since the writing thread will keep the region alive as long as the reading thread needs it alive, DecrThreadCnt operation, together with the corresponding IncrThreadCnt when the thread is created.

Another optimization applies when a goroutine 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 optimized away. Unfortunately, this optimization and the previous one exclude each other; we cannot apply both, even if both are otherwise applicable. If we did, we would optimize away a single increment of the counter but two decrements of that counter, leaving an incorrect final value.

When a thread `t1` sends a message to another thread `t2`, with a statement such as `send` v_1 `on` v_2, the code of 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 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 goroutine 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 $\mathsf{R}(c_1) = \mathsf{R}(c_2)$.

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

Table 1. Information about our benchmark programs

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 2. Benchmark results

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 goroutine 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.

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 1 has some background information about our benchmark programs. Some of these are adaptations of Debian's "Computer Language Benchmarks Game" provided by the GCC 4.6.0 Go test-suite and aimed at 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" [12], and the remaining programs are from libraries: Michal Derkacz's `blas_d` and `blas_s` [6], Dmitry Chestnykh's `passwordhash` and `pbkdf2` [5], and Andre Moraes' `gocask` [13]. 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 these allocations request. 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 times 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 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 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.

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, basically 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 `gccgo` runtime in Ubuntu's libgo0 4.6.1 provides a basic 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.

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 difference in MaxRSS 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 only increase code size, 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 GC versions of the benchmarks use one data structure that can suffer from internal fragmentation, while the RBMM versions use two.

The MaxRSS results for the benchmarks in the third group show that if a program makes extensive 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 fives 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* come be due to those 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. We have some ideas for optimisations that can reduce this overhead.

6. Related Work

Tofte and Talpin [16] introduced RBMM for Standard ML. 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 garbage collection was often faster. Aiken, Fähndrich and Levien [1] observed that significantly better results were possible by liberating region lifetimes from having to coincide with lexical scope, that is, from the stack discipline.

Cherem and Rugina [4] implemented RBMM for Java using a points-to analysis. The authors found that short-lived regions help memory utilization, and they found that many allocations could be placed on the program's stack.

Phan [14] also used a points-to-graph to implement RBMM for Mercury. Like Cherem and Rugina, he provided a flow-insensitive analysis and avoided imposing stack discipline on regions. His analysis created regions based on types, allowing different parts of a composite object to be stored in different regions. This can improve memory reuse [14], since it permits, for example, a relatively short-lived list skeleton to be stored in separate region from the list elements, which may have longer lives.

Berger, Zorn, and McKinley [2] 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. They also presented a new system, called a reap allocator, which acts as a combination of both general purpose and region allocators. Their results (Figures 5a and 5b) show that on both memory consumption and execution time, the custom region allocators beat the Doug Lea allocator on 3 of 5 benchmarks, and the reap allocator on 4 of 5 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.

Lattner and Adve [10, 11] used regions for C/LLVM. They found that 25 of the 27 benchmarks they tested ran faster using RBMM, or "pooled allocation", than using `malloc`. Grossman *et al*. [9] introduced Cyclone, a safe dialect of C. Unlike Lattner and Adve's system, Cyclone requires programmers to explicitly write calls to all the region operations: creation, allocation, and removal.

Gay and Aiken [7] investigated the use of regions in C@, their dialect of C. They counted references from pointers to determine when a region can be deleted. Gay and Aiken found maintaining these counts to be expensive. In contrast, our use of protection counts is much cheaper, since the counts need to be updated only at call sites, rather than at every pointer assignment.

Gerakios, Papaspyrou and Sagonas [8] proposed the use of a tree-based hierarchy of regions for parallelization. Their regions contained locks protecting the critical sections from parallel access. Later the authors implemented concurrent regions in Cyclone.

Boyapati *et al*. [3] proposed a memory management technique that combines regions and ownership types. This, like the Gerakios *et al*. approach [8], is safe from dangling memory pointers. Their implementation focused on a real-time version of Java.

7. Conclusion

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. Previous work has shown region-based memory management to be competitive with garbage collection in many environments. While our implementation is a work in progress, preliminary testing gives us hope that region-based memory management will also work well for Go.

Traditional region analysis algorithms propagate 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 likewise. 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 reanalyse the functions in the call chain(s) leading down to it.

We are in the process of extending our implementation to support all of Go. This means handling both parallel constructs (along the lines shown in Section 4.5) and higher-order constructs (including defer statements and interface types). We also intend to implement a number of optimizations, including multiple specialization of functions, as well as allowing different parts of a data structure to be stored in different regions if they have different lifetimes. While making these changes, we intend to ensure that reanalysis times remain practical.

References

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

[2] E. D. Berger, B. G. Zorn, and K. S. McKinley. Reconsidering custom memory allocation. In *OOPSLA 2002*, pages 1–12. ACM Press, 2002.

[3] C. Boyapati, A. Salcianu, W. Beebee Jr., and M. Rinard. Ownership types for safe region-based memory management in real-time Java. In *Proc. PLDI 2003*, pages 324–337. ACM Press, 2003.

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

[5] D. Chestnykh. Passwordhash and PBKDF2 Go libraries. URL `https://github.com/dhcest/passwordhash`.

[6] M. Derkacz. BLAS: Basic linear algebra subprograms for Go. URL `https://github.com/ziutek/blas`.

[7] D. Gay and A. Aiken. Language support for regions. In *Proc. PLDI 2001*, pages 70–80. ACM, 2001.

[8] P. Gerakios, N. Papaspyrou, and K. Sagonas. A concurrent language with a uniform treatment of regions and locks. In *Electronic Proceedings in Theoretical Computer Science*, pages 79–93, 2010.

[9] D. Grossman, G. Morrisett, T. Jim, M. Hicks, Y. Wang, and J. Cheney. Region-based memory management in Cyclone. In *Proc. PLDI 2002*, pages 282–293. ACM Press, 2002.

[10] C. Lattner and V. Adve. Automatic pool allocation for disjoint data structures. *SIGPLAN Notices*, 38:13–24, 2003.

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

[12] H. Li. Programming language benchmarks. URL `http://attractivechaos.github.com/plb/`.

[13] A. Moraes. Gocask library. URL `http://code.google.com/p/gocask`.

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

[15] Q. Phan, Z. Somogyi, and G. Janssens. Runtime support for region-based memory management. In *Proc. 8th ISMM*, pages 61–70. ACM Press, 2008.

[16] M. Tofte and J.-P. 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.

[17] M. Tofte and J.-P. Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, 1997.