Towards Region-Based Memory Management for Go

Matt Davis
Peter Schachte, Zoltan Somogyi, and Harald Søndergaard
mattdavis9@gmail.com, {schachte,zs,harald}@unimelb.edu.au

Department of Computing and Information Systems and NICTA Victoria Laboratories
The University of Melbourne, Victoria 3010, Australia

June 16, 2012

THE UNIVERSITY OF MELBOURNE

Google's Go programming language

- Go is a system-level programming language.
- Functions may be passed as arguments to functions (higher-order).
- To ensure memory safety, the language disallows pointer arithmetic, but requires runtime array bounds checking.
- Memory management is automatic (currently using garbage collection).
- Parallelization is made simple and safe via Go-routines (co-routines).

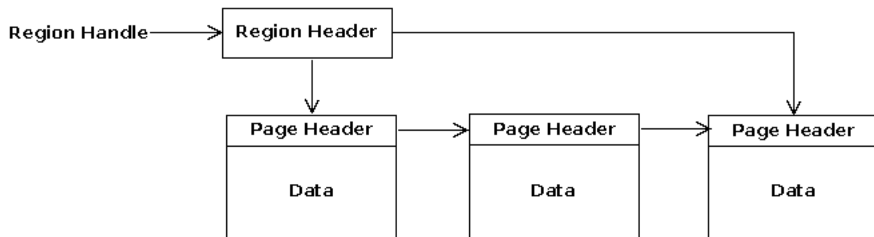
The problem

- Garbage collected systems automatically manage a program's memory at runtime, but require additional runtime resources.
- Garbage collection works by scanning the memory allocated for objects in the program. It identifies objects that are no longer needed and can have their memory reclaimed.
- The memory scan has to process each allocated object individually. If there are many such objects, this can be significant overhead.
- The memory scan is often implemented by pausing the execution of the program so that no objects can be updated during the scan. This pause negatively affects the perceived runtime performance of the program.

Region-Based Memory Management (RBMM)

- RBMM systems divide memory into regions.
- Each allocation takes place from a specific region.
- Objects are never deallocated individually. Instead, the unit of deallocation is a whole region.
- Static analysis puts two objects into the same region if their lifetimes end at the same point in the program.
- The compiler decides where a region should be created and reclaimed.

Regions visualised



The benefits of RBMM

- Reclaiming memory is fast, since many objects are removed at once, simply by updating a few pointers.
- Since the compiler decides where a region should be reclaimed, no scanning of live data is needed.
- Having objects with similar lifetimes grouped together has the potential to enhance cache locality.

The drawbacks of RBMM

- Object lifetimes can be undecidable, therefore a conservative static analysis must be used to approximate lifetimes.
- An object may be kept around longer than it needs to, because some other objects in its region are still alive.
- RBMM inserts additional function calls into the program; this adds runtime overhead.
- The added functions will increase the file size of the transformed binary.

Code transformation: region operations

The compiler inserts calls to basic region operations into the program:

- *CreateRegion* creates a contiguous area of memory for objects to be allocated from.
- *AllocateFromRegion* returns some memory from the given region for an object to use. Regions can expand if more memory is needed.
- *RemoveRegion* reclaims all the memory associated with a region (if allowed; discussed later).

The guiding principles of the program transformation:

- Create regions as late as possible.
- Remove regions as soon as possible.

Region creation and allocation

Our analysis detects the use of a region that has not been created, and inserts a region creation operation into the code.

Before:

```
func foo() {
  a := new(T)
  bar(a)
}
```

After:

```
func foo() {
  reg1 := CreateRegion()
  a := AllocFromRegion(reg1, sizeof(T))
  bar(a)
  RemoveRegion(reg1)
}
```

Function prototype and function calls

- If a function returns data that it allocates inside its body, a region must be passed down to it as an additional input argument.
- We transform function calls, so as to pass needed regions as extra arguments.

Before:

```
func bar(x *T) {
  x.next = new(T)
  baz()
}

func foo() {
  a := new(T)
  bar(a)
}
```

After:

```
func bar(x *T, reg *Region) {
  x.next = AllocFromRegion(reg, sizeof(T))
  baz()
  RemoveRegion(reg)
}

func foo() {
  reg1 := CreateRegion()
  a := AllocFromRegion(reg1, sizeof(T))
  bar(a, reg1)
  RemoveRegion(reg1)
}
```

Protection counting

The protection counter allows for the removal of regions at the earliest possible time, while also preventing premature region reclamation.

- We create regions and only pass them downwards from caller to callee.
- The earliest time to reclaim the region might be in a callee.
- If the caller needs a region after the call, it increments the protection count on the region before the call, and decrements it after the call.
- The *RemoveRegion* operation cannot reclaim a region whose protection count is non-zero.
- The protection counter is not an ordinary reference counter.

Protection counting: example

Before:

```
func bar(x *T) {
  x.next = new(T)
}

func foo() {
  a := new(T)
  bar(a)
  baz(a)
}
```

After:

```
func bar(x *T, reg *Region) {
  x.next = AllocFromRegion(reg, sizeof(T))
  RemoveRegion(reg)
}

func foo() {
  reg1 := CreateRegion()
  a := AllocFromRegion(reg1, sizeof(T))
  IncrProtection(reg1)
  bar(a, reg1)
  DecrProtection(reg1)
  baz(a, reg1)
}
```

Our implementation

We implemented our program transformation, and the program analyses it needs, in the *gccgo* compiler.

Our implementation uses plugins. Plugins provide compiler features that can be shared amongst users, without having to modify the actual compiler.

- Plugins avoid having to spend time rebuilding the entire compiler.
- A plugin can operate on the language agnostic middle-end intermediate language, *GIMPLE*, or the lower-level representation in *RTL*.

Region types

We introduce two region types:

- *Local Regions* are created and reclaimed dynamically.
- The *Global Region* is a single region holding allocated data for objects whose lifetimes are uncertain, such as global variables. The data allocated from this region is garbage collected using Go's usual garbage collector.

Multiple modules

A Go program can reference data from other Go libraries (object files). These libraries might be compiled with or without a region enabled compiler.

Solution:

- We preserve the results of our region analysis in each object file.
- Functions compiled with non-region-aware compilers do not expect the extra region parameters that a call to that function from a region-aware caller would pass.
- They also would not obey the invariants required by our system.
- Therefore our system always allocates data that is passed to non-region-aware modules from the Global Region.

Current status

- We have a basic version of RBMM working on a subset of Go.
  - We have designed the necessary support for go-routines, but have not implemented it yet.
- We handle higher order function arguments (with limitations).
- We handle multiple translation units (multiple modules).
- Our initial benchmark tests look promising.

Experimental results

Benchmark	MaxRSS (megabytes)			Time (secs)		
Name	GC	RBMM		GC	RBMM	
bt-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%)
bt	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%)

Future work

- Support the rest of Go, including go-routines.
- Modify our analysis and transformation to permit different parts of the same structure with different lifetimes to reside in separate regions.
- Optimize both the runtime and the code generated by our transformations.

Questions...

Questions?

Backup slides

Backup slides

Information about the benchmarks

Benchmark			GC			RBMM		
Name	LOC	Repeat	Alloc	Mem	Collections	Regions	Alloc%	Mem%
bt-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%
bt	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%

Experimental setup

- 30 runs of each benchmark application.
- libgo support from Ubuntu 11.10 for gcc 4.6.1
- Quadcore Intel i7-2600 (Dell Optiplex 990)
- 8GB Memory

History of RBMM

- Tofte and Talpin: ML implementation of RBMM (1994). This is considered the seminal work for RBMM.
- Gay and Aiken: C@ C-dialect supporting RBMM (2001).
- Grossman, Morriset, Jim, Hicks, Wang, and Cheney: Cyclone a safe-dialect of C (2002).
- Lattner and Adve: Pooled memory for LLVM (2002).
- Cherem and Rugina: Java implementation of RBMM (2004).
- Phan: Mercury implementation of RBMM (2009).

Why globals are evil (for RBMM)

- Global data is garbage collected, which is not always a cheap process.
- There are cases where regions are merged with the Global region. This creates a memory leak because the non-Global region being merged, allocated data from a different allocator. The garbage collector cannot reclaim data allocated by a different allocator.