Region Based Memory Management for Go

Matt Davis

Computer Science Department
University of Melbourne

October 28, 2011

1. Overview
2. Who am I?
3. Research
   - GCC Plugins
   - Go
4. What is Region Based Memory Management
   - Example
5. Progress
6. Future
7. Questions

Who am I?

Matt Davis

- PhD Researcher.
- Computer Science and Modeling/Simulation experience.
- In the Computer Science department at the University of Melbourne.

Research

Research: Region Based Memory Management

- Researching an alternative to garbage collection in the Google Go language.
- Using region based memory management to show application speedup, and efficient memory usage.
- Creating a GCC compiler plugin to demonstrate the region-based alternative to garbage collection for the Go language.
- Goals
  - Improve application performance by using region based memory management.
  - Learn about efficient memory allocation strategies.

GCC Plugins

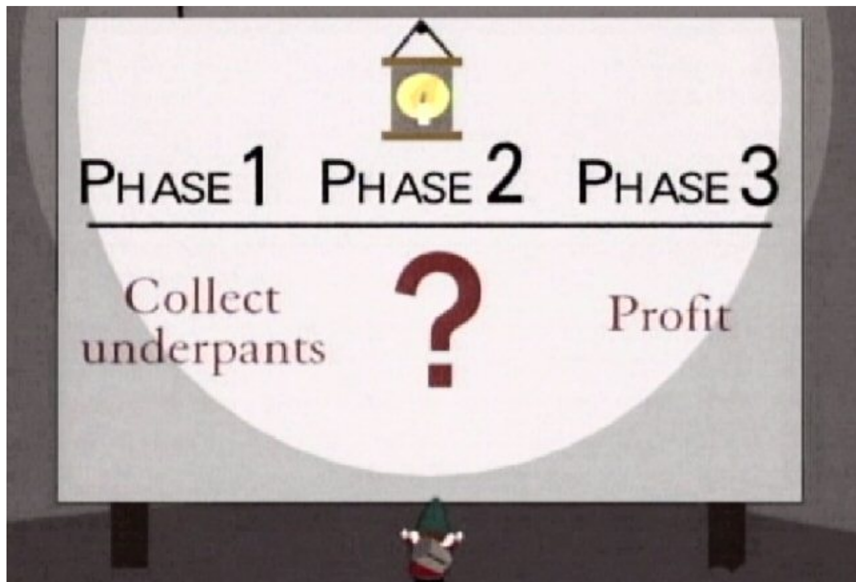
GCC Plugins

- Allows one to develop for the GCC compiler.
- Avoids having to rebuild the entire compiler.
- Can operate on the language agnostic middle-end intermediate language: *GIMPLE*.
- Can create shareable optimizations and analysis plugins.

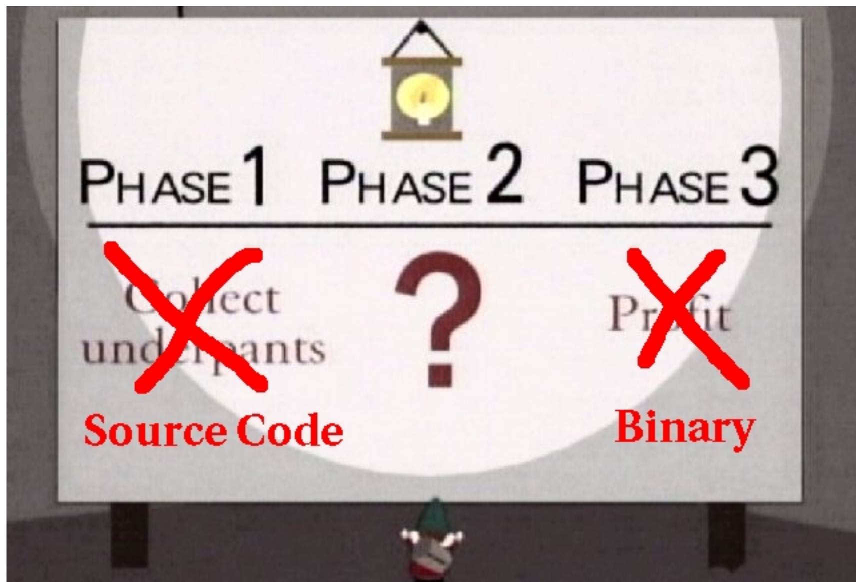
Google's Go Programming Language

- New language: 2009.
- Allows passing of functions as data (higher-order).
- Designed to be a system-level language (like C/C++).
- Provides memory safety (bounds checking) and defers (exceptions).
- Garbage collected memory.
- Disallows pointer arithmetic.
- Easy parallelization via Go-routines (co-routines).

What is Region Based Memory Management?



What is Region Based Memory Management?



What is Region Based Memory Management?

What is Region Based Memory Management?

- RBMM aims to improve program performance by making allocation and deallocation of dynamic memory fast.
- RBMM can also improve performance by enhancing cache locality.
- The compiler detects from static-analysis where data allocations occur.
- The compiler figures out when data reclaims (*free*) should occur.
- That data can then grouped based on object lifetimes in to regions of allocated memory.
- When the last piece of data in that region is no longer needed, the entire region and all of the objects in it is reclaimed.
- Objects can also be grouped by data type.

Program Heap Illustration

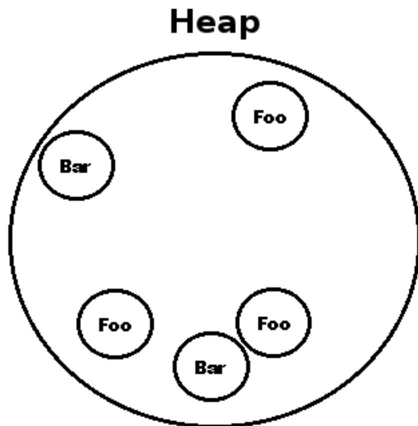


Figure: Traditional view of a program's memory space

Program Heap Illustration

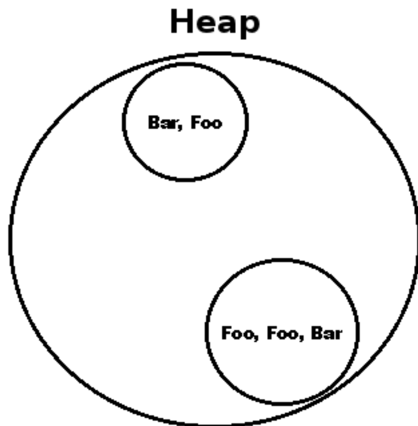


Figure: Region memory view of a program's memory space

Example Before RBMM Transformation

```
package main

type Thing struct {id int; next *Thing}

func dosomething(x *Thing) {
    x.next = new(Thing)
    x.next.id = 2
}

func main() {
    a := new(Thing)
    dosomething(a)
}
```

Example After RBMM Transformation

```
package main

type Thing struct {id int; next *Thing}

func dosomething(x *Thing, r *Region)
    x.next = __ALLOC_FROM_REGION(r)
    x.next.id = 2
}

func main() {
    r := __CREATE_REGION()
    a := __ALLOC_FROM_REGION(r)
    dosomething(a, r)
    __REMOVE_REGION(r)
}
```

Current Status

- Have a basic version of RBMM implemented working.
  - Does not work on higher order constructs.
  - Does not work for parallelization.
- Handles multiple compilation translation units (multiple modules).
- Initial benchmark tests look promising.
  - Worst case for garbage collection, a tree-data structure with many nodes. Our solution is faster than the garbage collected version and has a smaller peak virtual memory footprint size.

Future

Future

- Handle higher order functions.
    - Difficult because static analysis cannot guarantee what function is being called. The compiler has no idea what regions are needed at compile time.
- Implement safe parallelization.
    - How do we pass or manage regions to prevent concurrent access?

Questions...

Questions?