Precise Garbage Collection in C

PANKHURI

February 16, 2011

Agenda

- Problem Statement.
- Precise / Conservative Garbage Collection.
- What will Magpie do ?
- C semantics and Precise GC.
- When will Magpie fail ?
- Design and Implementation.
- Results.
- Conclusion.

GC Algorithms

- Conservative
  - Numerical Values interpreted as Live Pointers.
  - Mistake dead pointers as root leaving objects in heap indefinitely.
- Precise
  - Requires exact information about whether a given word is a pointer?
  - Counts references toward reachability only for words:
    - Whose static type is a pointer type.
    - Variables that are in scope.

C Semantics and Precise GC

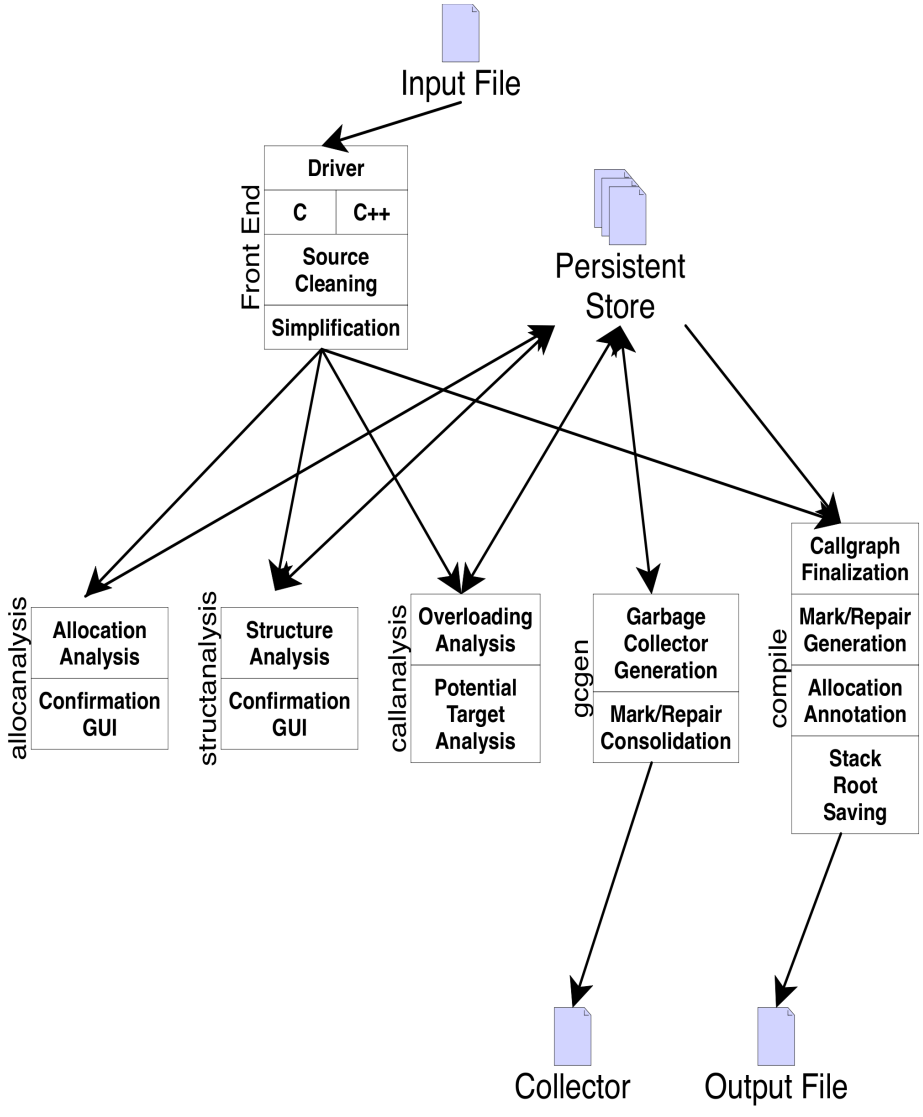
- Pointer Manipulation.
  - Pointer arithmetic.
  - Save Pointers?
  - Integer == Pointers ? Pointers == Integer
- Unconverted Libraries.
  - Will functions save refrences?
  - callbacks ?
- Explicit Deallocation.

```
s->p = unconverted_malloc();
...
unconverted_free(s->p);
do_some_allocating_work();
```

What information GC needs?

- Which words in the heap are root references?
- Where references exist in each kind of object?
- What kind of object each object in the heap is?

Design


Input File
Front End
Driver
C
C++
Source Cleaning
Simplification
Persistent Store
allocanalysis
Allocation Analysis
Confirmation GUI
structanalysis
Structure Analysis
Confirmation GUI
callanalysis
Overloading Analysis
Potential Target Analysis
gcgen
Garbage Collector Generation
Mark/Repair Consolidation
compile
Callgraph Finalization
Mark/Repair Generation
Allocation Annotation
Stack Root Saving
Collector
Output File

Allocation Analysis

- Determines what kind of object each allocation point creates.
- Magpie uses this information to tag allocated objects as having a particular type.
- This tag is used by mark and repair functions to generate appropriate traversal functions.

For example, the allocations

```
p = (int*)malloc(sizeof(int)*n);
a = (int**)malloc(sizeof(int*)*m);
```

are converted to

```
p = (int*)GC_malloc(sizeof(int)*n, gc_atomic_tag);
a = (int**)GC_malloc(sizeof(int*)*m, gc_array_tag);
```

Structure Analysis

- Determines which words in an object kind are pointers.
- Magpie uses this information to generate traversal functions.
- The mark function is used to traverse a structure or array during the mark phase of a collection.
- Repair function is used to update pointers when objects are moved.

Example:

```
struct a {
        int* x;
        int* y;
    };
```

```
Void gc_mark_struct_a(void* x_) {
struct a* tmp = (struct a*) x_;
GC_mark(tmp->x);
GC_mark(tmp->y);
}
```

```
void gc_repair_struct_a(void*x_) {
struct a* tmp = (struct a*) x_;
GC_repair(&tmp->x);
GC_repair(&tmp->y);
}
```

Call graph analysis

- Generates a conservative approximation of what functions each function calls.
- Magpie uses this information to eliminate roots in the local stack.

Tracking local variables

- Identifies the pointers on the stack and communicates to garbage collector.
- It performs this communication by generating code to create shadow stack frames on the C stack.
- Collector traverses them to find the pointers in the normal C stack.
- There are 4 kinds of frame supported by Magpie:
  - Simple Frame
  - Array Frame
  - Tagged Frame
  - Complex Frame

Stack Frames

Simple Frames:

prev. frame
length + bits 00
var/field address
var/field address
var/field address
...

Array Frames:

prev. frame
length + bits 01
start address
length
start address
length
...

Tagged Frames:

prev. frame
length + bits 10
start address
tag
start address
tag
...

Complex Frames:

prev. frame
length + bits 11
start address
traverser address
info
info
...

Example

Example:

```
// ORIGINAL
int cheeseburger(int* x) {
  add_cheese(x);
  return x[17];
}

// TRANSFORMED
int cheeseburger(int* x) {
  void* gc_stack_frame[3];
  /* chain to previous frame: */
  void* last_stack_frame = GC_last_stack_frame();
  gc_stack_frame[0] = last_stack_frame;
  /* number of elements + shape category: */
  gc_stack_frame[1] = (1 << 2) + GC_POINTER_TYPE;
  /* variable address: */
  gc_stack_frame[2] = &x;
  /* install frame: */
  GC_set_stack_frame(gc_stack_frame);
  add_cheese(x);
  /* restore old GC frame */
  GC_set_stack_frame(last_stack_frame);
  return x[17];
}
```

Handling Unions

- When a type contains a union of pointer and non-pointer types,GC need to follow and update a pointer variant only when it is active.
- The active variant of a union is tracked using an extra byte outside of the object.
- This byte is set whenever a field of the union is assigned or its address is taken.
- Mark and repair functions consult the byte to determine whether to follow or repair the pointer variant.

```
union {
int i;
int* p;
} a;
```

```
a.i = 1;
iwork(&a);
a.p = q;
pwork(&a);
```

```
a.i = 1;
GC_autotag_union(&a, 0);
iwork(&a);
a.p = q;
GC_autotag_union(&a, 1);
pwork(&a);
```

Results

Experience with PLT Scheme.

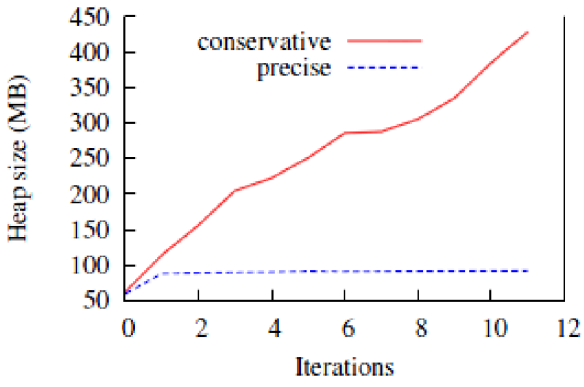


Figure 2. Running DrScheme inside DrScheme

Performance :

1) For benchmarks that spend relatively little time collecting, the conservative GC can be up to 20% faster than precise GC. This is mostly due to the overhead of registering stack-based pointers and other cooperation with the GC.

2) Programs that allocate significantly tend to run faster with precise GC, due to its lower allocation overhead and faster generational-collection cycles.

3) Thus, the benchmarks illustrate that we pay a price in base performance when building on a C infrastructure with precise GC compared to using conservative GC.

- Experience with C programs
- Experience with Linux Kernel.

Conclusion

- In most cases, Magpie performs within 20 percent (faster or slower) than the original C code .
- Requires no more effort than the existing Conservative collector.
- Removes memory spikes created by Conservative GC.