Precise Garbage Collection in C

Pankhuri

February 16, 2011

1 What is this paper about ?

Magpie is a source-to-source transformation for C programs that enables precise garbage collection, where precise means that integers are not confused with pointers and the liveness of a pointer is apparent at the source level. It serves as a bridge between C and C++ and precise garbage collection. It works by gathering sufficient information for the collector via static analyses and queries to the programmer. This information is transferred to the runtime by translating the original source code.

2 Precise / Conservative Garbage Collection.

1. **Conservative Garbage Collection:** It is a style of garbage collection that treats the stack as an ordinary range of memory, and assumes that every word on the stack is a pointer into the heap. Thus, the collector marks all objects whose addresses appear anywhere in the stack, without knowing for sure how that word is meant to be interpreted.
   (a) **Problems:**
      - Numerical values to be interpreted as live pointers to otherwise dead objects, causing memory leaks.
      - Conservative collectors may mistake dead pointers as roots, leaving objects in the heap indefinitely.
2. **Precise Garbage Collection:** It is a style of garbage collection that requires exact information about whether or not a given word in the heap is a pointer or not. It counts references toward reachability only for words that correspond to values whose static type is a pointer type, and only for words that correspond to variables that are in scope.

3 C Semantics and Garbage Collection

The key constraint a C program must obey to use precise GC is that the static types in the program source must match the programs run-time behavior,at least to the extent that the types distinguish pointers from non-pointers.

Major Concerns:

3.1 Differentiate Pointers / Non - Pointers

1. For precise GC, the values of variables and expressions typed as non-pointers must never be used as pointers to GC-allocated objects, because objects are not considered to be reachable through such references.
2. In addition, to support GC strategies that relocate allocated objects, Magpie disallows references through non-pointer types even if the same object is reachable through a pointer-typed reference.
3. GC implementations that relocate objects require that a pointer-typed object either refer to a GC-allocated object or refer outside the region of GC-allocated objects; otherwise, an arbitrary value might happen to overlap with the GC-allocation region and get moved by a collection.
4. Supports Interior Pointers.
5. Supports Unions and (Void*).

3.2 Connect allocation with shape of allocated data.

1. Magpie requires that the types of data allocated by a C program fit into four categories: atomic(non-pointer) data, arrays of pointers, single structures, and arrays of structures.
2. A combination of type and category is determined from each allocation expression, which must use a recognized allocation function, typically malloc, calloc, and realloc, but Magpie supports a configurable set of allocators.

sizeof(t)	***: allocates a single t structure***
sizeof(t) * e	***: allocates an array of t structures***
sizeof(t*) * e	***: allocates a pointer array***
e	***: allocates an atomic block***

3.3 When Magpie Fails?

1. **Pointer Manipulation:** Pointer arithmetic can shift an address so that it does not refer to an allocated object, and then further arithmetic can shift the address back. If a collection can occur between the arithmetic operations, then a reachability assumption of our GC has been violated.

   ```
   int* p = GC_malloc(sizeof(int) * 1024); p -= 1024; work(p); p[1024];
   ```

2. Saving Pointers to disk and later dereferencing them.
3. Using exclusive-or operations to collapse pointers in a doubly-linked list.
4. Pointers as Integers and Integers as Pointers.
   - Integer in a pointer type location?
     (a) It GC supports relocation then program may change the value stored at that pointer.

    (b) If GC does not support relocation then it may cause an object which should be freed to be retained.

  - Pointer stored in an Integer location?

    (a) Program may crash as the memory could be garbage collected anytime.

5. **Unconverted Libraries:**
   - A library function might keep a pointer that it was given and try to use the pointer in later calls. Memory could be garbage collected second time it is referenced leading to program crash.
   - A library function might invoke a callback that uses precise GC before the library function returns. When a library invokes callbacks into code that uses GC, then any objects originally passed to the function may have moved by the time the callback returns.
   - **Solution for above problems can be :**

     (a) Have programmers tag the functions calls which save pointers.

     (b) Convert the libraries also.

6. **Explicit Deallocation:** An explicit non-GC deallocation might release a page of memory back to the underlying allocator, and the GC might later acquire that page. If the deallocated reference is not cleared, then it may point into the GC-allocation area without pointing to an allocated object.

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

7. If unconverted free () releases a page of memory, and if the GC starts using the same page during do some allocating work (), then s's reference to p is left pointing into the GCable region and probably not to the beginning of an object. If the GC does not allow interior pointers to GCable objects, the program can crash.

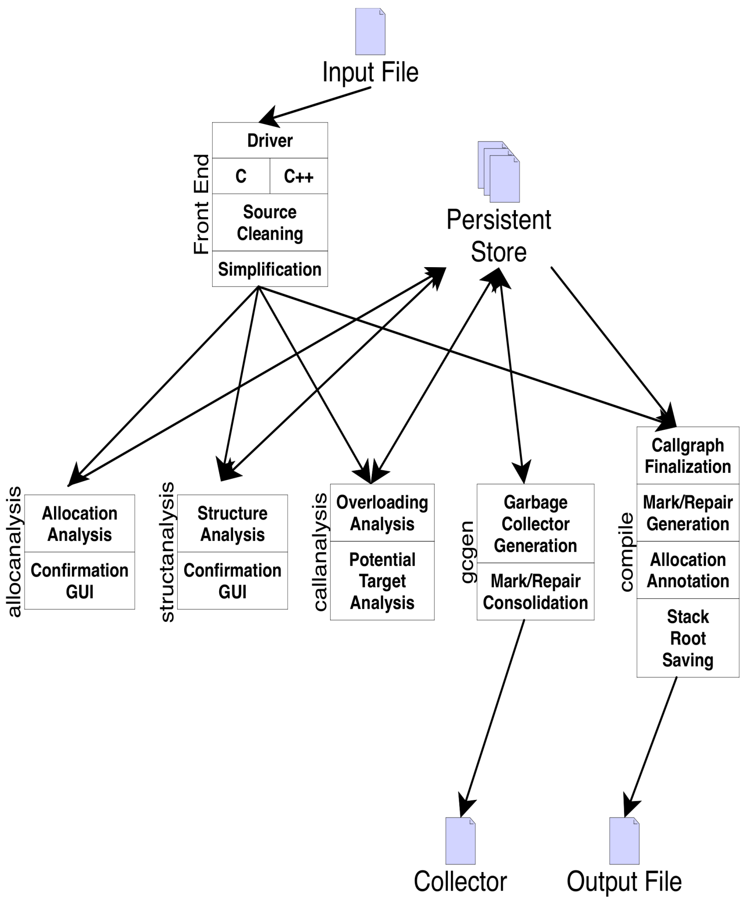
4 Transformation

4.1 Requirements for Garbage Collection

1. Which words in the heap are root references?
2. Where references exist in each kind of object?
3. What kind of object each object in the heap is?

After Collector has gathered above information it iterates through the list of root references, marking the referenced objects as it goes. For each object in the set of marked objects, propagation works by looking up what kind of object that object is and then invoking the appropriate traversal function on it. When a fix point is reached (no more objects have been added to the set of marked objects), any unmarked object is deallocated.

4.2 Design



4.3 Phases of Magpie

1. **Allocation Analysis:** The 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);

2. **Structure Analysis:** The 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, while the repair function is used to update pointers when objects are moved.

Example:

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

3. **Call Graph Analysis:** The call graph analysis generates a conservative approximation of what functions each function calls. Magpie uses this information to eliminate roots in the local stack. If a program calls an unknown function, Magpie must assume that the function may trigger a garbage collection. Thus, it must convert the source to notify the collector of any live local pointer variables for the collector to use them as roots. If, however, the call graph analysis has determined that the call will not lead to a garbage collection, it does not need to transfer this information to the collector.

4. **Tracking Local Variables:** This phase is one of the most crucial phases of Magpie it 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, which the collector can then traverse to find the pointers in the normal C stack.

   There are four types of stack frames used in Magpie:

   - Simple Frames
   - Array Frames
   - Tagged Frames
   - Complex Frames

   All these frames have first two words in common. First word points to the previous frame. The second word is separated into 2 parts. Last two bits of this word indicate the type of frame while rest of the bits is used as length field. In simple frames, the length field gives the total size of the frame. For array and tagged frames, the length refers to the number of arrays or tagged items in the frame. For complex frames, the length refers to the number of informational words appended to the end of the frame. In all cases but complex frames, the frames are capable of storing information about more than one stack-bound variable at a time.This merging avoids as much space overhead as possible.

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:

```
// 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];
}
```

To decrease the number of variable information saved Magpie uses Call, Liveliness and Initialization optimization.

4.4 Handling Unions:

When a type contains a union of pointer and non-pointer types, the mark and repair functions 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 whenever a field of the union is assigned or its address is taken. The automatically generated 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);
```

In addition, active-variant information must be copied between unions on struct assignment and struct copying via memcpy.

4.5 Handling Global and Static Variables:

Magpie locates all static/global variables and registers them in initialization functions that are called when the program starts. Each variables location is registered with a tag that provides mark and repair functions and, if necessary, union variant tracking.

5 Implementation and Experience

5.1 PLT Scheme.

Experience with PLT Scheme.

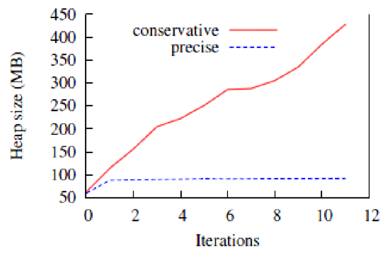


Figure 2. Running DrScheme inside DrScheme

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.

5.2 Experience with C programs.

Unlike PLT scheme which was developed in a way that constraints imposed by precise GC was followed, normal c programs are expected to violate the assumptions made by Precise GC.Results for C programs showed that precise GC is a viable alternative to manual memory management for a typical C program. However, it is not especially beneficial for short-running programs without untrusted components.

5.3 Experience with Linux Kernel.

In this environment, the benefits of precise GC are less clear, for two reasons. First, the user/kernel interface of an OS tends to be much narrower than a high-level language interface,directly preventing user-mode code from causing space leaks in the kernel. Second, low-level concurrency and hardware-level interactions can create problems for a precise collector.Precise GC is less obviously desirable in these settings.

6 Conclusion

In most cases, Magpie performs within 20 percent (faster or slower) than the original C code and requires no more effort than the existing Conservative collector. The memory use of Magpie-converted code similarly tracks the Conservative collector, and in some cases removes memory spikes created by Conservative GC. According to me Magpie is beneficial for application which follows constraints imposed by precise collection. It can also help in solving memory issues for long running applications but for short programs and applications with many third party plugins it does not provide substantial benefit.