Garbage Collection - Part 1: Overview, allocation, reference counting

2009-03-25 / rev 1 / lhansen@adobe.com

This note provides an introduction to the GC subsystem and covers allocation of garbage collected and reference counted storage. The files covered are GCObject.{h,cpp}, GCAlloc.{h,cpp}, and GCLargeAlloc.{h,cpp}. The rest of the collection algorithms and data structures are covered in Part 2. Various gripes ar collected in Part 3.

The main user documentation for MMGC is at Mozilla Developer Center: https://developer.mozilla.org/en/MMgc

Major findings

As was the case for FixedMalloc, the amount of wasted space in objects ("internal fragmentation") is entirely hidden because size classes are chosen to pad out a block, not to fit a particular kind of object demographic. This fact means that we've no good grip on whether the object management mechanisms are efficient. The large number of segregated allocators is probably a poor fit for small heaps but we can't know that until we've measured internal fragmentation.

Overview of automatically managed memory

Requirements

So far as I can figure, the following services and facilities are required from the garbage collector subsystem.

- Reasonably fast and memory-efficient allocation of object memory.
- Objects must fit in with the C++ type systems (eg, GC types can be subclassed in C++).
- C++ finalizers must be accomodated.
- Automatic reclamation of all unreferenced allocated objects.
- Prompt automatic reclamation of client objects that might have external visibility (eg, windows).
- Balance memory consumption with performance.
- An object can be referenced (for liveness purposes) by a tagged or untagged pointer and by pointers embedded at arbitrary aligned locations in memory known to the GC, including the program stack.
- There must be a facility for weakly held objects (objects that may be reclaimed if referenced only from specially designated "weak" locations).
- It must be possible to explicitly deallocate GC-managed objects.
- The collector must preserve interactivity, eg, collection pauses must be short. (The requirements are soft.)
- Debugging facilities (memory profiling, interior pointers, non-incremental collection, ...?)

There is no form of requirement for multiple thread support, indeed GC facilities must only ever be invoked by a single program thread. (Note that this is more strict than mutually exclusive access from multiple threads, which is not a requirement.)

General Description

GCObject is a lightweight base class for managed objects that don't require finalization, while GCFinalizedObject is the same for objects that must be finalized. RCObject is a subclass of GCFinalizedObject that provides reference counting.

A multi-pronged management strategy has been adopted. First, objects can be deallocated explicitly, and this helps take pressure off automatic memory management. Second, a large number of types in the VM and in the Player are reference counted types. Deferred reference counting as used in AVM will tend to deallocate free structures relatively quickly, thereby lifting the burden otherwise placed on the whole-heap garbage collector and in general reducing memory consumption. Finally, the garbage collector reclaims circular reference-counted structures and other data.

The combined effect of explicit deallocation, reference counting, and garbage collection is seen in the String class, which is reference counted but which references garbage-collected string data. The string data are deallocated explicitly by the String object's finalizer if the data are known not to be shared. When the string data are shared, however, the garbage collector is responsible for reclaiming them.

Deferred reference counting operates by maintaining a set (the zero-count table, ZCT) of objects whose reference counts have reached zero. Once this set is full, it is reaped: the program stack is scanned and elements in the ZCT that are referenced from the stack are preserved, while the others are reclaimed. The amount of free space in the ZCT is kept relatively small, so the effect is to run fairly frequent "minor" garbage collections, which keeps the pressure off the more expensive "major" collections.

The true garbage collector is triggered by allocation volume and heap expansion, or explicitly by the user program. It traces out the graph of live objects from a set of roots, and reclaims all non-live objects. Tracing is conservative: anything that looks like a pointer to an address that is known to hold an object will keep that object alive. Once tracing is complete, the heap can be swept and non-live objects can be reclaimed.

The main negative consequences of conservative tracing are to make tracing more expensive; to make it hard to move objects, as the collector can't know whether a value is a pointer or something that just looks like one; and to retain objects that are falsely believed to be referenced because a value that isn't a pointer looks like one.

Conservative tracing is attractive because it does not require a cooperating compiler or programming language: a pointer need not be specially marked as such, but can be stored anywhere in memory that the collector knows to examine for pointers.

Since the heap can grow very large the collector is incremental: objects to be traced are pushed onto a stack, and some tracing is performed every time the collector allocates a block of memory. This interleaves marking and program execution, but at a cost: when a pointer is created from one object to another while marking is in progress, the store must be recorded so that the pointed-to object can be traced. The write barrier that performs the recording slows down the user program.

Issue: The nomenclature in the GC code is mixed: the mark stack is indeed a stack, but the GC code uses the word "queue" in many contexts.

Action: Needless confusion ensues; clean it up. Low priority.

Incremental collection does not prevent visible pauses, because some activities at the beginning and end of a collection cycle are executed in an atomic fashion and depend on the amount of work to be

done as well as on uncontrollable user code.

Issue: Synchronous execution of finalizers is brittle in several ways: they can cause visible pauses, and there are restrictions on what they can do, in particular comments in the code state that they may not allocate garbage collected storage.

Action: Not a short-term project, but we should consider ways of moving finalizers out of the GC's critical section.

Garbage collectable objects (GCObject.{cpp,h})

GCObject and GCFinalizedObject

Objects derived from GCObject or GCFinalizedObject are reclaimed automatically if no references to them are found during a collection cycle. Objects derived from GCFinalizedObject have a (virtual) destructor and that the destructor (the object's *finalizer*) is called when the object is about to be destroyed.

Issue: There are undocumented classes whose uses are unknown and which are effectively unused in the AVM: *GCFinalizable*, *GCFinalizedObjectOptIn*. There are no references to them from the Flair checkout that I have either.

Action: Document & justify those classes.

GCObject adds no storage overhead to its subclasses; it merely provides *new* and *delete* operators and a method for obtaining an object's weak pointer (which is stored externally to the object, see the section on GCWeakRef).

GCFinalizedObject adds a vtable pointer, of course.

RCObject

Reference counted objects (*RCObject* and its subclasses) are objects that are reclaimed if their reference counts reach zero and they are not referenced from memory that pins them (the program stack and certain other memory). RC objects are always finalizable, and they are garbage collected too: if an RC object is unreachable without its reference count being zero, it is still deleted. (This can happen as the result of bugs, circular program structures, or reference count overflow.)

Reference counting is primarily useful if the reference counted data structures are not circular. In AVM, the class *ScriptObject* extends RCObject, and ScriptObject is in turn the base class for a large number of classes in the VM and in the avmglue in the Player. (RCObject is the direct base class for a number of other classes in the Player, and for the *String* class in AVM.) Indeed it could seem that reference counted objects dominate in the heap.

Issue: What are the demographics of the heap in terms of RCObject / GCObject?

Action: We need object demographics from real applications.

Issue: I am aware of no evidence that the RCObject-derived classes are generally part of noncircular data structures and that they benefit from being RCObjects. Benefits would appear in the form of reduced time spent on memory management and space saved compared to garbage collection. In particular, we need to be sure that *on balance* it is a win for ScriptObject to be derived from RCObject and not from GCObject or GCFinalizedObject.

Action: This is very hard to test; we should do nothing until we have data about demographics, at least.

An RC object requres a field for the reference count and other book-keeping data. In RCObject there is a four-byte field *composite* -- a manually managed bit vector -- that holds this information. Since RCObject derives from GCFinalizedObject which adds a one-word vtable pointer, the *composite* field probably does not incur any padding overhead.

Issue: A comment in the code states: "*b/c RCObjects have a vtable we know composite isn't the first 4 byte and thus won't be trampled by the freelist*". This is false; the vtable need not be situated at the beginning of the object at all, so user fields can very well be situated there.

Action: This portability problem should be cleaned up, if the code still depends on it, or at a minimum a compile-time assertion should be added that ensures that there is not a problem. (It might take a while to track this kind of bug down during porting.)

The reference count value is only eight bits, but there is a test for overflow; if the field overflows a flag is set that makes the reference count sticky, and the object has to be reaped by the general garbage collector.

Block storage and the page map

The GC maintains a set of blocks from which objects (large and small) are allocated. It needs to know whether a value might be interpreted as a pointer into one of the blocks it maintains; this is particularly crucial for object tracing during garbage collection. For this purpose, it maintains a page map: a bit vector with two bits per page, covering the range of memory between the lowest and highest block addresses.

The page map is an integral number of blocks; it can grow but not shrink. On a 32-bit system the largest page map (covering 4GB) would be 256KB. A single-page (4KB) page map is sufficient for contiguous heaps up to 64MB.

Object allocators

The memory manager divides allocations among a number of allocators for small objects of a given size and a separate allocator for large objects. The small objects are segregated not just by size, but by whether they are atomic (contain no pointers), are reference counted, or are general GC objects. Since there are 40 size classes, a total of 120 small-object allocators are created.

Issue: No evidence is presented that this organization is good for small-memory systems. It is probably good for performance, though it's not clear to me (yet) why reference counted objects are segregated from general objects.

(One could hypothesize that since reference counts are updated frequently the segregation would tend to concentrate writes on certain pages. This might matter on VM systems; it probably does not matter much on small systems, not even for cache locality.)

Action: Figure out whether this division of size classes makes sense for small systems and whether there are benefits to combining RC and non-RC storage.

Small objects (GCAlloc.{cpp,h})

The class *GCAlloc* implements an allocator for a fixed-size GC managed object. (The GC class allocates a number of these.) Object sizes are always divisible by eight. Objects cannot exceed the size of a block (again 4KB, there are four 0xFFF constants in the code that appears to be a reflection of this, so consider the 4KB hardwired) minus a *GCBlock* header, apparently 12 words.

Blocks are allocated via the GC class, not directly from GCHeap. The GC class needs to know about blocks that participate in GC management of objects.

Each block has a bit vector holding four bits per object: marked, queued, finalizable, and weakly-held. These are uses by the GC. The bit vector storage is discussed in the next section.

The segregated header storage introduces some complexity, eg, multiplication by variable offsets is needed to compute the bit position from an object offset in the block, slowing down access to the bits. The benefit is in reducing storage overhead: four bits per object rather than (typically) eight bytes for a traditional object header (one word for the data and one for padding).

The bit vector is either stored inside the block (if the block has pointer-containing data *and* there is space for the bit vector) or in a separate bit vector storage area, which has a memory manager of its own.

For non-pointer-containing data the justification for storing the bit vector off-page is that it improves performance in virtual memory systems - pages containing these data will have to be read, presumably, but won't ever have to be written back because we treat them as read-only.

Issue: The utility of that optimization on both modern desktop systems (where page caches are gigantic and virtual memory is not a bottleneck for most users) and on non-virtual memory systems is highly uncertain. There's a comment from Tommy to that effect as well.

Action: Consider simplifying this by removing the off-page bit vector storage. Low priority, probably.

The off-block bit vector manager has one free list per size class and carves bit vectors off pages allocated from GCHeap. Pages are never returned to GCHeap, and free items are never coalesced.

Issue: There is a risk that the off-block bit vector manager will use more memory than it ought to. Suppose we have a program that allocates 1MB of objects of one size class, frees them all, then 1MB of another size class, and so on. It never uses more than 1MB of data memory but the bit vector storage will grow each time objects of a new size class are allocated.

Action: We need data about how much memory this memory manager uses in practice, on realistic programs.

Each block has a free list of objects in the block as well as a pointer to free storage at the end of the block. The allocation path seems fairly long:

- check that there is a current block (and create one if not);
- check whether the block needs sweeping (and sweep it if so);
- pick an object off the free list or carve it off the end;
- compute the bit vector index of the block's gc bits;
- clear the block's gc bits;
- set the appropriate gc bits;
- perform accounting;
- check whether the block was exhausted (and make it not-current if so);
- check whether we're collecting (and mark the object if we're in the middle of a collection).

Issue: Is the cost of allocation too much? The path is long and is reached through several layers of calls.

Action: Unclear that efficiency is crucial at this point. We probably want actionable data.

Issue: There is no way of telling the difference between free memory, overhead memory, or block-internal waste memory.

Action: Implement this; see similar notes in FixedAlloc review.

Large objects (GCLargeAlloc.{cpp,h})

This is an allocator for objects larger than those that fall into the memory manager's size classes (currently the largest non-large object is 1968 bytes). The objects are allocated in whole blocks from the underlying GC instance; eventually the request ends up as a block allocation request in GCHeap.

Unlike the case for FixedMalloc the blocks are not pristine; a small header is allocated at the start of the first block. This means that user programs that have a fondness for object sizes divisible by 4KB will allocate one block more than expected, most of which will be wasted.

Issue: The allocator sets the block's *usableSize* to the allocated size minus the header size, not to the requested size. So internal fragmentation goes unrecorded.

Action: Fix that, we want to know how much there is.

Issue: On GCLargeAlloc::Free, the list of all large blocks must be walked to find the block. If there are more than a few, and a fair amount of churn, then this is just slower than it needs to be - and has worse locality. Other parts of the GC appear to pay attention to VM locality, so why not here?

Action: Consider fixing it; at least try to figure out if it's a problem in real programs.

Deferred reference counting

Reference counting has two main aspects, reference count maintenance and management of objects whose reference counts are zero.

Reference count maintenance

The main issue with reference maintenance is usually its cost (execution time and code size). Consider *IncrementRef*:

```
void IncrementRef()
{
  if(Sticky() || composite == 0)
    return;
  composite++;
  if((composite&RCBITS) == RCBITS)
    composite |= STICKYFLAG;
  else if(InZCT())
    GC::GetGC(this)->RemoveFromZCT(this);
}
```

(Recall that *composite* is a word-size collection of bit fields.) This isn't too bad but it's a bit branchy and could account for a fair amount of code if it's heavily in-lined. The *sticky* flag isn't used much elsewhere and is slightly redundant with a zero reference count field. If the RC field were to be moved

into bits 23-31, and the high bit made into an all-purpose "slow path" flag (that doubles as *sticky*, probably), and *composite* made signed, then overflow from the reference count would automatically set the slow path flag. Then the entire function body could look like this:

```
if (composite < 0)
  IncrementRefSlow();
else
  composite += (1 << 23);
```

Issue: How often is IncrementRef inlined? How often is it called and how performance critical is it? (If it's not performance critical it shouldn't be in-line. And only the performance-critical parts should be in-line.)

Issue: In general it seems to me that there are special cases in reference count management that should probably be handled elsewhere, for example the Pin method checks that the object is not deleted because pinning a deleted object will cause crashes later. But this could be avoided if there were a 'deleted' bit that did not depend on *composite* being zero.

Issue: DecrementRef also looks a little complicated.

Action: Over time the RC logic could probably be streamlined some.

The ZCT

When the reference count on an object reaches zero, DecrementRef adds it to the "zero count table", ZCT. Periodically (not less often than every 512 adds) this table is reaped. The reaper deletes all objects that are in the ZCT that are not referenced from the program stack and other such protected areas.

Issue: The ZCT may grow without bound; so far as I can figure it is only limited by the size of the program stack. It never shrinks. Recursively deleting a large reference-counted list could make it large, but in practice I expect it to be small.

Action: It would be good to verify that it stays small.

ZCT reaping can be triggered explicitly, and the Player does this. Comments in the player code state that it calls ReapZCT because Collect (the full garbage collector) is slow on large heaps.