Garbage collection - Part 2: Collection

2009-03-27 / rev 1 / lhansen@adobe.com

This note covers garbage collection, weak references, and write barriers. The files covered are GC.{h,cpp}, GCWeakRef.h, GCStack.h, and WriteBarrier.h. Requirements, introductory matters, objects allocators, and reference counting are covered in Part 1. Various gripes are collected in Part 3.

Major findings

Mark stack management does not seem like it is a good fit to small-memory systems: failure to extend the mark stack will cause an abort, and the mark stack is maintained as one large and therefore fragmentation-inducing block.

The worst-case marker recursion depth -- 87KB -- is far too large for embedded systems.

Since GC::Collect is called by the player to drive the collector and does not have any way of using a quantum of time provided to it, we risk visible pauses, especially on systems where the heap/CPU ratio is high (as it will be on phones).

The code implementing the policy for triggering and running the garbage collector seems quite shaky, and may not have seen a lot of testing (since the Player drives GC quite a bit and DRC takes pressure off the GC). The fact that the various GCs are unaware of each other and largely independent of the FixedMalloc heap also suggests that the collection policy is not a good fit for the smaller systems.

I believe the collector design has too many restrictions to be truly resilient to out-of-memory conditions.

Incremental collection

All collection is incremental.

("Non-incremental" collection is supported but it is implemented simply as the incremental collector run until collection is finished. The logic for triggering collections in non-incremental mode is a little different but not very. Non-incremental mode is used for debugging only.)

Collection can be triggered and driven by client code, or it will be triggered and driven as a side effect of allocation. Either way things progress pretty much the same way.

The collector proper has five main functions:

- *StartIncrementalMark* finishes any outstanding lazy sweeping and pushes all the roots onto the mark stack and calls IncrementalMark
- *IncrementalMark* processes items on the mark stack until it's empty or until the time budget for the increment is exhausted. If the mark stack is empty on entry to IncrementalMark then it calls FinishIncrementalMark.
- *FinishIncrementalMark* marks from all the roots again, then creates a work item for the entire stack and marks that, and then calls Sweep
- *Sweep* calls Finalize and then sweeps any entirely empty pages
- *Finalize* calls the Finalize method on all the allocators owned by the GC; this causes each to traverse its block list and run the finalizer of each unmarked finalizable object, and compute the

number of live items per block. Blocks without live objects are added to lists that are processed by Sweep

Non-empty pages are not swept during collection, but on demand as more memory is needed, and sometimes eagerly before a new collection starts.

Since marking and sweeping are interleaved with mutator work, the main latency in the collection process comes in at the beginning, when StartIncrementalMark sweeps unswept pages, and at the end, when FinishIncrementalMark scans the stack and examines every page in the system and runs the finalizers for all objects that are going to be reclaimed (through GC::Finalize).

Marking, Finalization, and Sweeping

Conservative marking

MMgc uses conservative marking. Any pointer-size value, read from a location with pointer-size alignment within a root or object in the garbage-collected heap, that can be interpreted as a pointer to an object within the garbage-collected heap, causes the latter object to be marked.

Conservative marking tends to be more expensive than exact marking (due to a more complicated test); to retain more objects than exact marking (due to false positives); and to prevent objects from being moved (since the GC does not know whether a value is a true pointer it can't update any value) . The latter two factors tend to drive up memory consumption relative to exact marking.

The memory consumption problem can be alleviated to some extent. MMgc segregates pointer-containing and non-pointer-containing objects, and considers only pointers near the start of an object as being a pointer to the object - not arbitrary pointers into the object. Both techniques are known to help.

Issue: MMgc does not implement other techniques, like blacklisting.

Action: Research such techniques and find out if they apply to MMgc. (Blacklisting might not, on non-VM systems, for example.)

Issue: It would be helpful to try to estimate whether conservative marking retains more memory in practice than exact marking would, especially as false retention tends to be platform-dependent (it depends on the location of objects in memory).

Action: It's not clear how that could be done without implementing exact marking.

Issue: In GC::MarkItem, the case for small objects handles doubly-inherited classes specially but the case for large objects does not. I'm not sure I understand what's going on for small objects later, presumably the C compiler has given us a pointer into an object following a cast to a base or derived class. The code that's here is a hack and probably not portable. Presumably the code is missing for large objects.

Action: Figure out what's supposed to be the case here.

Mark algorithm

Marking (the core of which is GC::MarkItem) proceeds as follows. First root objects are pushed onto the mark stack. Then the marker loops: it pops an element off the stack, marks it, and scans it for pointers, pushing each pointed-to object onto the mark stack. The algorithm terminates when the stack is empty.

If the pointing object and the pointed-to object are on the same 4KB page then the marker calls itself recursively instead of pushing the item onto the mark stack. No justification is given for that, presumably it is meant as an optimization.

Issue: The largest number of items on a page is 506 eight-byte items. In principle, a Lisp-style linked list of 506 GCObjects could all end up on the same page. Compiled by GCC on my Macintosh, the stack frame for MarkItem is 172 bytes. So the worst-case recursion for MarkItem consumes 87KB of stack - far beyond the 50KB stack size currently required to be shared by the VM, the player, the player's host, and the GC. (I stress that this worst case is extremely unlikely to occur and will therefore be encountered but rarely in web content.)

Action: This must be fixed. Two easy fixes would be not to go recursive at all, or to limit recursion to some small number of items. It seems that the first order of business must be to determine whether the recursive marking pays off significantly on real programs.

Mark stack

GCStack (see GCStack.h) is a simple mark stack used by the collector to track objects that have been seen but not traversed.

The mark stack storage is allocated with *new*, which -- because of operator overloading -- allocates it on the FixedMalloc heap. (There's a NULL check following the allocation; it is redundant but also ineffective, because it would not prevent an almost immediate crash in GCStack::Push when GCStack::Alloc returns.)

Issue: Because FixedAlloc may abort the process if memory cannot be found, mark stack allocation may cause an abort. This is unacceptable, as the GC must be able to run to completion in low-memory conditions.

Action: The incremental marking mechanism must in some way be able to handle a failed memory allocation. There are known algorithms for this; they trade speed for space. Segmenting the stack may be helpful too, as an initial segment will be all that's needed for such a fallback algorithm.

The stack size does not appear to be bounded. It starts out at 512 items, and forcing it to grow to 1024 and then 2048 was not very hard. 2048 elements represents 16KB, since the element type is GCWorkItem and that type is 8 bytes on a 32-bit platform.

Issue: The mark stack is one block of memory; if it grows much the large blocks will tend to induce fragmentation, so the fuller the heap the less likely that we will complete.

Action: At least allocate the mark stack in segments.

Issue: The mark stack does not shrink, so if it needs to be large during a particular GC cycle all the memory will be retained until the process shuts down.

Action: Allocating the mark stack in segments will make it easier to free unused parts.

Sweeping

The process of scanning a block or the heap looking for reclaimable objects, finalizing them, and recycling the memory is known as sweeping.

The sweeper is run in three places.

- During StartIncrementalMark, any pages not yet swept are swept. This sweep collects free objects on a page into the page's free list.
- When a new block is needed for one of the allocators, it is taken off the list of blocks with free objects, and swept if necessary; this sweep also collects free objects on a page into the page's free list.
- During FinishIncrementalMark, the heap is swept from one end to the other. This sweep is different: it runs Finalize (see the next section), then frees any completely empty blocks. Non-empty blocks are added to free lists. But no sweeping of individual blocks takes place.

The sweeper incorporates an optimization to avoid work for blocks that haven't changed since the last GC.

The general idea about running the sweeper lazily is to reduce the latency at the end of a garbage collection.

Issue: Since finalization touches every page anyway, do we really gain much by sweeping lazily?

Action: As always, we need data. This particular item is probably low priority.

Finalization

Finalization runs the finalizers of all dead, finalizable objects. To do this, it examines the bit vector of every heap object to find dead objects, so it touches every block in the heap (even if only to retrieve the address of the bit vector). Blocks are reached in more or less arbitrary order.

Issue: Finalization may be very slow and may result in visible pauses. The finalization phase may have dubious locality, each finalizer can take arbitrarily long to run, there can be lots of finalizers (every reference counted object is finalized), and every finalizable dead object is finalized at the end of the collection. Slow computers with small caches and slow memories trying to perform "web browsing" tasks may be particularly vulnerable.

On the other hand, we don't know whether many of the RC objects aren't simply taken care of by ZCT reaping (after all, that's the hypothesis for DRC), so whether this is a bottleneck in practice remains to be seen.

Action: "Follow up on this" is vague but that's what we need to do. Do we have *any* data on latencies in typical applications? Should I be worried that the new flying-cubes Flash animation on adobe.com has a consistent stutter as the "H" glides into place?

The finalization pass collects those blocks that are entirely empty so that they can be freed immediately by the sweeper.

Collection triggering

Triggering of garbage collection can be explicit (through calls to a GC API function) or implicit (as a consequence of allocation).

The Player is quite active in controlling garbage collection, and this probably affects (a) any policy parameters the GC itself has for running the collection and (b) the reliability of the GC triggering code in MMgc, because it's probably not been tested extensively.

In addition to triggering collections, the Player also triggers ZCT reaping by calling GC::ReapZCT; as explained elsewhere this is a "minor garbage collection".

Issue: The code that performs collection decisions is duplicated throughout the collector, and with various variations (sometimes a number is divided by the free space divisor and compared to a third; other times the third is multiplied by the free space divisor and compared to the first), and the effect is to cause confusion. (Since I'm on my soap box, *collectThreshold* is a poor name for the minimum heap size, since "threshold" could be an upper or lower limit.)

Action: Clean this up.

Explicit collection requests

The two APIs for triggering a collection are GC::Collect and GC::MaybeGC.

GC::Collect starts a full mark-sweep collection provided the final phase of a collection (reaping or sweeping) is not currently underway. Also, this function will drive an incremental collection that is underway to its conclusion.

GC::MaybeGC either starts a new incremental collection or pushes incremental marking along a little bit (which may also trigger the end of the collection).

Issue: Neither the VM nor the Player uses GC::MaybeGC, and there's no selftest for it. We must assume it's not been tested at all.

Action: Test it or remove it.

Issue: Since the player calls GC::Collect from time to time there's reason to be nervous about long pauses in programs with large heaps and/or on small devices.

Action: A better API would seem to be one that takes an argument that indicates the time available, and which may start a new collection and then push an existing collection along for that time.

GC::MaybeGC will only start a new collection if "enough" time has passed since the previous collection ended, and the memory consumption makes it appear that it's worthwhile.

Issue: The definition of "enough" time is in terms of "ticks", the units of which are machine dependent. The definition looks broken because it is constant and not parameterized by the machine. It is this (the comment is from the code):

```
kMarkSweepBurstTicks = 1515909; /* 200 ms on a 2ghz machine */
```

On a machine with a 1KHz tick frequency this is 1515 seconds, or 25 minutes. If the frequency is 1Mhz, 1.5 seconds. Machine speed does not seem to have anything to do with it.

This constant is also used for triggering automatic collections.

Action: Fix it.

Automatic collection

Automatic collection is driven by block allocation requests: as long as allocation stays within the existing allocated blocks and free lists, no new collection will be started automatically.

Issue: Though correct in principle, that policy is probably not ideal for small systems because it will tend to keep blocks alive longer; those blocks can then not be repurposed, eg for FixedMalloc.

Action: Investigate whether triggering off eg object allocation volume leads to lower block retention rates. (Object allocation indicates mutator activity, and with mutator activity objects tend to die.)

At the time of block allocation, provided that the entire heap (not just the gc'd heap) is large enough (1MB) and has been expanded since the last collection and enough time has passed since the last incremental collection completed, then a new collection is started.

Issue: That logic is a bit puzzling. It would seem to keep the incremental collector chugging along steadily, independent of actual expansion volume. (Just a single block expansion will trigger GC.)

But a collection can also be started if the gc'd heap is large enough (1MB as well) and "enough" block allocations have taken place since the last collection, where "enough" is defined as the number of GC pages divided by the free space divisor (hardwired at 4).

Issue: The *intent* of the free space divisor is not documented, as with too much else about MMgc. Given that block allocation updates both *allocsSinceCollect* and *totalGCPages* by the same amount when a block is allocated or freed, the test is that a>(t+a)/4, or a>t/3. I have no idea whether that was the intent, but I'm inclined to think not.

Action: The policy must be documented better and corrected if necessary. If we have tuning knobs, they must be predictable.

The manually managed heap (FixedMalloc et al) and the automatically managed heap (GC objects and RC objects) are only loosely coupled: the GC is aware of the "total heap size", which includes memory allocated to FixedMalloc, and uses lack of movement in the total heap size as a throttle on garbage collection, but that seems about it.

Issue: Yet pressure on FixedMalloc can sometimes be alleviated by running the garbage collector more often. This does not seem to happen at all, except accidentally - FixedMalloc does not seem to be aware of the GC.

Action: Investigate whether it's reasonable to make heap expansions in FixedMalloc trigger or drive collections in the GC'd heap(s).

There can be multiple garbage collectors, but they seem to be unaware of each other.

Issue: Yet pressure on one GC can sometimes be alleviated by running the other collectors more often. Again, this does not seem to happen at all except accidentally, by activity in a mostly idle heap being triggered by allocation in that heap, noticing that the heap has grown overall.

Action: Investigate whether it's reasonable to make heap expansions in one GC trigger or drive collections in the other GCs.

Collection progress

Once an incremental collection is in progress it is driven by block allocations: any block allocation will run the incremental marker, provided "enough" time has passed since the previous incremental mark.

The notion of "enough" time is given by the constant *kIncrementalMarkDelayTicks*; it is a number of ticks corresponding to 10ms:

```
kIncrementalMarkDelayTicks = int(10 * GC::GetPerformanceFrequency() / 1000);
```

Issue: There should be other drivers for collection as well, as outlined in the previous section: overall heap pressure should drive collection.

Action: Fix this.

Write barrier

Notionally a write barrier is a daemon that intercepts updates to the object graph and tracks them in some manner.

The write barrier is required for the incremental marking collector. During marking, an object's references are marked recursively, and once marked an object will not be revisited. However, if an assignment subsequently stores a pointer to an unmarked object into the marked object then the unmarked object will not be marked. The barrier records the store so that the marking can take place.

In MMgc a write barrier is implemented as a fat pointer with semantics attached to creation, assignment, and destruction. The write barrier is embedded in a GCObject/GCFinalizedObject (or any of its subclasses, including RCObject) and represents the reference from the embedding object to the referent.

There are two write barriers, one specialized to RCObject referents (*WriteBarrierRC*) and the other for non-RCObject objects (*WriteBarrier*). Tommy tells me that the split is a performance win; a general write barrier that figures out the type of object at run-time was too expensive. The issue is that WriteBarrierRC must do two things: it makes sure that the GC observes liveness, but it also keeps reference counts up to date.

Issue: The split is an unfortunate aspect with the current barrier structure, because it means that knowledge of the type of object is embedded in client code, which in turn makes changes in representation much harder. Notably, ScriptObject is derived from RCObject so every reference in the system is wrapped in a a WriteBarrierRC. We cannot experimentally change ScriptObject to be derived from GCFinalizedObject, which *otherwise* has the same semantics as RCObject without rewriting a lot of code (yet we'd like to do so because we want to find out if RCObject is a good or bad choice for ScriptObject).

Action: It's possible we can fix this -- for the purposes of an experiment -- by introducing a "joint" write barrier type and defining both DWBRC and DWB as this type, don't know yet.

The write barrier is fairly expensive. I've not included all the evidence here but looking at WriteBarrierRC, and making reasonable assumptions about function in-lining, a barrier costs at a minimum three function calls, a dozen ALU instructions, and several conditional branches before it is even determined whether it is necessary to record the store.

Issue: A lot of unnecessary work is performed for each store, only writes during incremental marking need incur that work.

Action: Consider testing whether the incremental marking is ongoing in-line, before doing all that work, then try to optimize the barrier by merging more of this functionality into a single function perhaps.

A conversation with Tommy about this reveals that so far, barriers have not shown up in profiles as "hot", and he speculates that that may be because there are so many other inefficiencies in the VM. Nevertheless it seems to me that we might look into reducing the cost in general, eg by building pathological test cases.

Weak references

GCWeakRef (see GCWeakRef.h) is a fat pointer that holds a pointer to a GCObject; the contained pointer is invisible to normal tracing, so if the only path to an object is via a GCWeakRef then the object will be a candidate for reclamation. The utility of the GCWeakRef is that a pointer to the GCWeakRef can be placed in a data structure, thus the weakly held object remains in the data structure as long as it is also held by some other reference, but once all other references are gone the reference from the data structure is dropped by the garbage collector.

A weakly held object is marked as weakly held (the kWeakRef bit is set for the object in the object's owning block's bit vector).

The garbage collector maintains a table of GCWeakRef instances, this table maintains a one-to-one mapping between the GCWeakRef and the object it holds. That mapping is needed when a weakly held object is reaped: at that point, the object pointer in the GCWeakRef must be set to NULL.

The hash table grows to maintain at most a 0.75 load factor. It grows by a factor of two when it does grow, and uses two words per entry. For a string of n insertions it will occupy at least 2*n*1.3 words plus some metadata. For 100 insertions, for example, it will occupy at least 260 words or about 1KB, on a 32-bit system.

Issue: The hash table for weakly held objects can become a large indigestible blob if there are many weakly held objects. And indeed, for mobile we are discussing adding a user-facing phantom pointer mechanisms, and that may increase pressure on the hash tables.

Action: At least try to get a grip on what the hash table size is at present (for typical Flex apps) and what it might be in the future - do we have hundreds of items or thousands? Also investigate alternative, less blobbish data structures. *GC::GetWeakRef* and *GC::ClearWeakRef* are probably not very performance critical, so something like a B-tree might work well in practice.

Issue: I believe there are no test cases for weak references.

Action: Implement such test cases. They must test that weakly held objects are indeed removed by GC.

OOM Handling and Collector Limitations

My concern here is whether the collector architecture prevents efficient out-of-memory recovery. This section is preliminary.

Recall that out-of-memory recovery must have several components:

- A callback to user code to allow user code to clear out caches, compress data structures, and so on
- A "phantom reference" mechanism to automate clearing of caches where suitable
- The ability to run a garbage collection following the callback to user code
- A reserve memory that's available for allocation during cleanup callback execution

I won't hold it against MMgc that there's no phantom reference mechanism at this time, nor a working reserve. However, we can imagine what such mechanisms might look like and question whether they will be usable.

Issue: User code cannot reliably allocate GC'd memory in a callback. GC'd memory may not be allocated from a finalizer. Yet for a phantom pointer mechanism the ability to record useful data about an object that's being cleared out may be necessary.

Issue: User code cannot reliably free GC'd memory explicitly. For example, Free() calls during sweep (from a callback, or from a finalizer) will be ignored.

Issue: A garbage collection cannot be triggered during a ZCT reap, yet a finalizer may need to allocate memory. Currently it can't, but as I wrote above that needs to be fixed probably, at which point this will become an issue.

Issue: As previously noted, the mark stack may need to grow during GC, which may be difficult or impossible if the system is in an out-of-memory situation. If "impossible" occurs, then the system will halt.

Efficiency

This section is preliminary.

As a general note, some of the algorithms are fairly branchy, because of optimizations and general complexity (eg, conservative marking has a complex predicate for pointer detection).

In general there is some inlining where I wouldn't expect it and lack of same where I would have expected it, but it's hard to be very critical without data.

At this time I have no major misgivings about the efficiency of the code. That said, the code is written to be fast, and making it more memory-efficient may impact performance on the micro-level as well as on the macro-level.