

Request Oriented Collector (ROC) Algorithm

Rick Hudson and Austin Clements

June 2016

golang.org/s/gctoc

We propose a new garbage collection (GC) algorithm based on the *request hypothesis*¹. Put simply, the request hypothesis states that objects created by a request tend to die once the request is fulfilled. Our new request oriented collector or ROC algorithm focuses attention on these objects thus improving overall GC throughput and scalability while delaying the need to do a full heap GC.

The request hypothesis grew out of our work supporting cloud applications which are typically architected to have a goroutine handle a request from the network or from another goroutine. In fulfilling such a request the goroutine receives a message, unmarshals it, performs a computation, marshals the result, and places the result on a channel or socket destined for another goroutine or the network. The goroutine then terminates or becomes dormant waiting for another request. The computation may access a large shared heap to read data, but typically writes very little to the heap that persists beyond the request. If at some point the goroutine does share newly allocated object with other goroutines, we call this *publishing* that object. If an object isn't published, we call it *local*. The key observation of ROC is: if a goroutine exits before publishing an object, the object becomes unreachable and the associated memory can be reclaimed and reused immediately. The ROC algorithm is a way to promptly and efficiently reclaim these unpublished objects.

Now that we understand the use case we will provide a high-level description of the ROC algorithm along with enough detail to convince the reader that the algorithm is correct and terminates.

The heap is divided into spans containing objects and associated metadata such as pointer/scalar maps and mark bits. Each GC cycle walks the heap marking reachable objects in a data structure called the mark bitmap. To allocate an object the mutator simply sweeps the mark bitmap until it encounters an unmarked bit. It then allocates the new object from the associated memory. Each span maintains a current sweep pointer demarcating mark bits that have been swept and mark bits yet to be swept. Each running goroutine maintains an initial sweep pointer that along with the current sweep pointer can be used to determine what objects it has recently allocated. Objects between the initial sweep pointer and the current sweep pointer were either marked reachable or have been allocated by this goroutine. Unmarked bits

¹ The term request oriented collector (ROC) replaces transaction oriented collector (TOC). This will remove the confusion between the use here and ACID property transactions familiar to the database community. Thanks to Randall Farmer for the suggestion.

between the initial sweep pointer and the current sweep pointer denote objects that have been allocated but not published.

The write barrier is responsible for maintaining the invariant that all published objects are marked. The write barrier is called whenever a pointer to an object is written into the slot of another object. The write barrier can determine whether an object is local or published. If the referrer object (the object holding the slot being written into) is published and the referent (the pointer being written into the slot) is local, the referent is about to be published. Furthermore, objects transitively reachable from the referent are also about to be published. To maintain the invariant that published objects have their mark bit set, the write barrier sets the mark bit of the referent and does a transitive walk of objects reachable from the referent setting mark bits. Only then is the object published by writing the referent into the slot. No synchronization is needed because local objects are only visible to the local goroutine and mark bits are set before this object is published. A branch of the transitive walk is terminated when it encounters a marked object. The local goroutine only has to scan local objects and local objects aren't being mutated during the scans.

This write barrier design allows ROC to efficiently reclaim unpublished objects when a goroutine exits simply by resetting the current sweep pointer back to the initial sweep pointer. Because unpublished objects do not have their mark bits set, this action is sufficient to free them and make them available for allocation to another goroutine, without having to wait for the next GC cycle.

Informal proof

We now informally outline a proof of correctness, completeness, and termination of the write barrier. There is a finite bounded number of local objects, each mark reduces the number of unmarked local objects, and any infinite structure must either be undergoing mutation, which it is not since the only mutator with access is in the write barrier, or contain a cycle of unmarked objects. Since we mark objects before we scan them we break any such cycle of unmarked reachable local objects. These facts can be used show correctness, completeness in the sense all published objects are marked, and termination. More importantly this can be used to conservatively identify unpublished objects without synchronization.

When a goroutine is started it has allocated no objects and thus has published no objects and the initial sweep pointer is the same as the current sweep pointer. All objects are allocated unpublished with unset mark bits and lay between the initial sweep pointer and the current sweep pointer. The write barrier maintains the invariant that published objects between the initial sweep pointer and the current sweep pointer will have their mark bit set. Once terminated a goroutine neither allocates nor publishes objects.

Examples

To better explain the algorithms let's use some old school ASCII art.

Figure 1

This is the typical state of a mark bitmap after a full GC cycle.

```
10011110010100101010100001001011010010110100101001011101010111101
^
Allocation Start
^
Current Sweep Pointer
```

1 indicates object was reachable at the last GC.
0 indicates the object is free and available for allocation.

Figure 2

This is the typical state of a mark bitmap after allocation *without* ROC. Allocation starts at the beginning of the span and proceeds to the end of the span. The boundary between allocated and unallocated areas of the span is demarcated by the Current Sweep Pointer.

```
10011110010100101010100001001011010010110100101001011101010111101
^                                <- before ^ after ->
Allocation Start                  Current Sweep Pointer
```

1 indicates object was reachable at the last GC.
Before: 0 allocated since the last GC
After: 1 object in use, 0 object free

Figure 3

This is a typical state of a ROC bitmap which tracks the per goroutine allocations. Note that the unmarked bits between the goroutine's initial sweep pointer and the current sweep pointer indicate newly allocated objects. Unlike Figure 1 we also maintain the current goroutine's Initial Sweep Pointer.

```
10011110010100101010100001001011010010110100101001011101010111101
      ^                                <- before ^ after ->
      goroutine                        Current Sweep Pointer
      Initial Sweep Pointer
```

1 indicates the object was either reachable at the last GC or was allocated and published since.

Before: 0 allocated since the last GC and not published.
 After: 0 object free

Figure 4

The algorithm ensures that published objects have their mark bit set.

The write barrier maintains this invariant by flipping unmarked bits to marked before publication

```
10011110010100101010101011101001011010010110100101001011101010111101
      ^      ***      <- before ^ after ->
      goroutine                               Current Sweep Pointer
      Initial Sweep Pointer
```

1 indicates the object was either reachable at the last GC or was allocated and published since.

Before: 0 allocated since the last GC and not published.

After: 0 object free

```
if referent is marked and referenced is unmarked {
  recursively mark referenced
  install reference in referent slot
}
```

*** These 3 objects have had their bits flipped by the write barrier as part of being published
 Unpublished newly allocated objects have a 0 mark bit.

Figure 5

At this point the request ends and the goroutine exits.

The algorithm resets the Current Sweep Pointer to the goroutine Initial Sweep Pointer

```
10011110010100101010101011101001011010010110100101001011101010111101
      <- before ^ -> after
      goroutine Initial Sweep Pointer
      ^
      Current Sweep Pointer
```

1 indicates the object was either reachable at the last GC or was allocated and published since.

Before: 0 allocated since the last GC and not published.

After: 0 object free

Note that the published objects have their mark bits set while the unpublished objects which are no longer reachable do not.

Figure 6

Instead of creating and exiting goroutines another typical Cloud architecture has goroutines that become dormant awaiting a new request. Typically such dormant goroutines have a shallow stack and few if any pointers to unpublished scratch objects. The algorithm deals with dormant goroutines by scanning their stacks and transitively marking any locally reachable objects.

```
100111100101001010101011101001111010110110100101001011101010111101
      ^               *      *      ^      -> after
      goroutine               Current Sweep Pointer
      Initial Sweep Pointer
```

1 indicates the object was either reachable at the last GC, was allocated and published since, or was allocated and is reachable from the stack of a dormant goroutine.

Before: 0 allocated since last GC, not published, and not reachable from goroutine stack

After: 0 object free

* Local objects reachable from dormant goroutine's stack are marked

Figure 7

At this point the request becomes dormant but objects reachable from the stack, if any, are marked.

The algorithm resets the Current Sweep Pointer to the goroutine Initial Sweep Pointer allowing the unmarked objects to be reused.

```
10011110010100101010101110100101101001011010010100101001011101010111101
  <- before ^ -> after
              goroutine Initial Sweep Pointer
              ^
              Current Sweep Pointer
```

1 indicates the object was either reachable at the last GC, was allocated and published since, or was allocated and reachable from the stack of a dormant goroutine.

Before: 0 allocated since last GC, not published, and not reachable from goroutine stack

After: 0 object free

At some point most of the bits become marked and we need to run a full GC cycle.

Next Steps

We are comfortable with the algorithm and the proofs so the next step is to implement the algorithm and make it available for testing and measurement. Optimizations will be driven by an iterative build and measure process that will include multiple groups. Based on the feedback and the performance numbers a decision will be made whether the algorithm will be included as part of the standard Go release.