

Upgrading Ruby's Garbage Collector

Doug Beaver - seattle.rb - 6/28/2005

Mark/Sweep GC

- Ruby uses a mark/sweep garbage collector
- At GC time, all active nodes are marked
- Once marking is done, sweep phase occurs
- The heap is walked sequentially, and any nodes not marked are freed

Ruby's Mark/Sweep Implementation

- Active nodes are walked
- Each active node has the mark bit set in its internal flags field
- At sweep time, nodes without mark bits are freed, and active nodes have their mark bit cleared (in order to be ready for the next GC run)

Why is the current mechanism bad?

- The current mechanism is simple and easy to use
- The simplicity comes at a price!
- Since all active nodes are being changed during the mark phase, copy-on-write (COW) semantics are destroyed
- This is bad news for ruby code that uses `fork()` (most famous is Rails/FastCGI)

How do we fix this?

- We need to find a way to annotate which nodes are in use
- But not touch the nodes themselves when we do it
- We also don't want to change the speed of the GC too much

Solution: Mark Table

- We create a mark table that holds the addresses of marked nodes, and instead of changing the nodes themselves when marking them, we update entries in the mark table
- This is relatively easy, since `st.c` in the Ruby distribution gives us a nice low-level tree implementation

Change details

- Added the mark table
- Added functions to check/set the mark table (mark_node and is_node_marked)
- All checks against mark flags in gc.c were moved to use the new functions
- At the end of the sweep phase, the mark table is flushed so it's empty for the next GC run

Status

- Mark phase runs okay, but it's not clear that it's marking all active nodes that it should be
- Sweep phase sometimes run into bus errors when trying to free nodes
- The major code changes have been made, so now it's a matter of squashing bugs and polishing things up
- Extensive debug logging calls are in the code, so I've been writing scripts to help me find bugs

Some characteristics of mark/sweep GC

- Since node relationships can be complex, you have to walk all nodes in the mark phase (you can't stop halfway through)
- Once a node is found to be unused, it is unlikely it will spring back into existence on a future run
- Fragmentation becomes a problem over time unless compacting is done

Possible future changes

- Sweep table
- Passive marking
- Compact phase
- Native thread GC

Sweep table

- Instead of incrementally freeing unused nodes at sweep time, create a sweep table to hold unused node addresses
- You then have the option of freeing all the unused nodes at the end of your sweep phase, or cutting short the sweep phase and doing incremental freeing of unused nodes at allocation time (lazy sweeping)

Sweep Table II

- With lazy sweeping, you can do constraint-based GC where you only sweep for N milliseconds, or until you free M KB of memory
- This is only possible because you're keeping track of unused nodes between GC runs
- Lazy sweeping lets you enforce limits on the amount of time you spend in GC, which opens the door to things like soft realtime scheduling

Lazy/Passive Marking

- Mark phase is the most expensive phase
- What if we could reduce the amount of time we spend marking nodes inside our GC run?
- If we passively mark nodes we know to be active, then we don't have to look at those nodes when we're inside our mark phase during GC

Passive Marking II

- We can't just lazily mark all nodes, since that would slow down non-GC runtime performance too much
- But we can find low-hanging fruit and mark those nodes
- Anything we can do to reduce time spent in the mark phase is good

Passive Marking III

- This is very complex, though!
- You have to cover cases like when a node is deleted or not being used anymore
- Still, it's worthy of exploration, there might be some techniques hiding in there that would allow us to reduce time spent in the mark phase

Compact phase

- You could periodically compact the heap whenever certain heuristics are hit
- This would require allocating a new heap, and copying the nodes over to it in packed order
- Once the new heap was created, the entire old heap could be destroyed and freed
- This is only good if fragmentation becomes a huge issue with your app workload

Native thread GC

- If native threads were added to Ruby, then we could have a separate GC thread that does incremental GC continuously
- This would avoid the spikes in CPU usage that you get with the synchronous mark/sweep GC that Ruby uses today
- Very difficult though, since you need to navigate locks and critical paths

Upgrade downsides

- Most of these new features require more tables, bitmaps, or extra heaps, which in turn require more memory
- If you're writing a GC, you typically want to keep allocations inside of GC runs to a minimum
- This can be addressed with careful coding, but it makes development of these features somewhat difficult

Metaruby to the rescue?

- Metaruby would make exploration of GC implementations many times easier
- You can't predict final performance using Metaruby's subset language, but you can predict relative performance, which might be good enough
- Performance is a concrete thing with GC, you have to pay careful attention to it since GC performance has a direct impact on perceived performance of applications

Contact Info

- doug.beaver@gmail.com
- seattle.rb mailing list: ruby@zenspider.com