Self-Collecting Mutators are Self-Compacting

An analysis of memory consumption in real-periodic programs

Stephanie Stroka

Embedded Software Engineering
University of Salzburg

January 25, 2011

Outline

The ideal allocator

- Fast memory allocation and deallocation
- No "wasted" memory space
  - Internal fragmentation
  - External fragmentation
- Predictable time and memory consumption
- Fast compile-time

The problem

- Fast allocation $\hookrightarrow$ wasting memory
- Less fragmentation $\hookrightarrow$ slow allocation

$$\boxed{OptimalMemoryConsumption \sim \frac{1}{AllocationTime}}$$

- Predictable memory consumption
  $\hookrightarrow$ unreasonable run-time (memory copying)
  $\hookrightarrow$ unreasonable compile-time (program analysis?!)

Typical program characteristics [1]

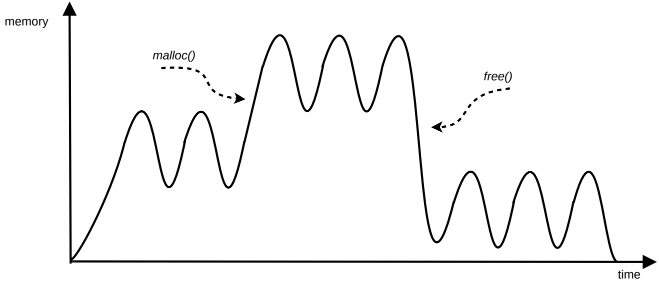

Figure 1: Periodic memory allocation and deallocation

Memory consumption in Garbage Collectors

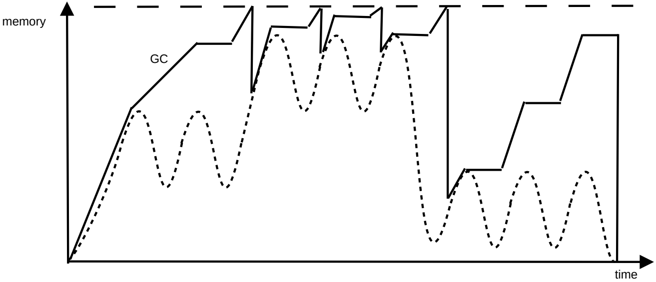

Figure 2: Garbage Collector loses periodic memory characteristics

Periodic memory allocation in the heap

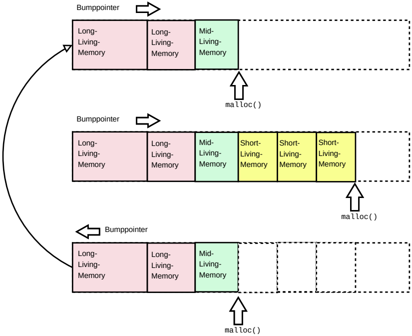

Figure 3: Periodic memory allocation and deallocation in the heap

Typical program characteristics [2]

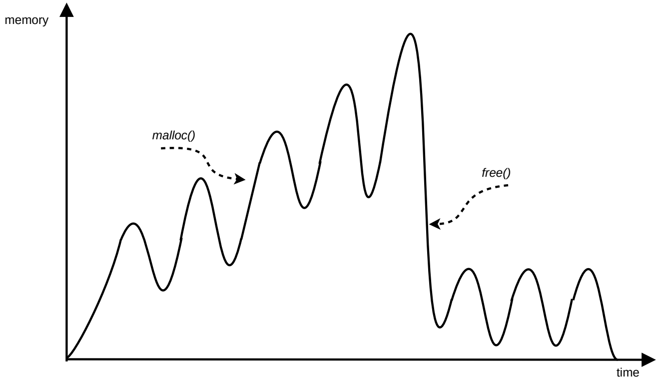

Figure 4: Periodic memory allocation with growing list

Semi-periodic memory allocation in the heap

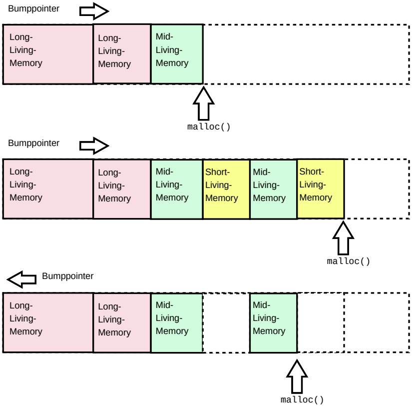

Figure 5: Periodic memory allocation with growing list in the heap

Analyzing techniques

To analyze periodic memory consumption, we need...

- an allocator that provides information about fragmentation
  $\hookrightarrow$ Compact-Fit
- a mutator that provides time information
  $\hookrightarrow$ Self-Collecting Mutator

Binding memory life-time to a clock

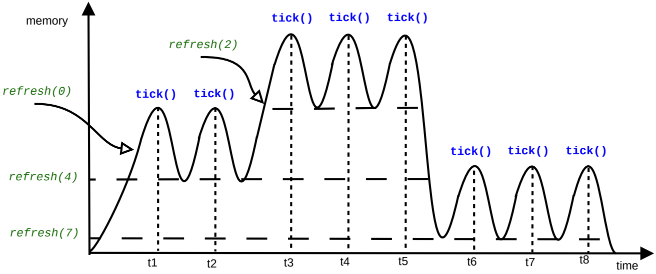

Figure 6: Periodic memory allocation with self-collecting mutators

Implementation

- SCM eager collection $\hookrightarrow$ all expired objects will be deallocated at `scm_tick()`
- SCM's malloc function uses `cf_malloc()` instead of `ptmalloc2()`
- Example program with `scm_malloc()`, `scm_refresh()` and `scm_tick()`

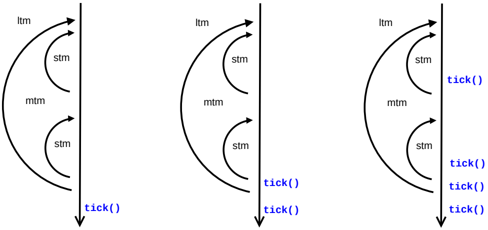

Figure 7: Analyzing memory consumption behaviour with growing #ticks

Results

- Runtime increases with growing # ticks
  ↪ `tick()`, `refresh()` and `compaction()`
- Memory consumption decreases with # ticks
  ↪ max. fragmentation = #`free()`
  ↪ ...unlikely, because of periodic characteristic
- Compaction causes compaction

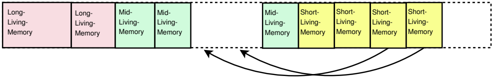

Figure 8: Compaction causes compaction

Results

- 1 tick
  - $\kappa = 1$, compaction = 160
  - $\kappa = 2$, compaction = 520
  - $\kappa = 3$, compaction = 0
  - $\kappa = 4$, compaction = 0
  - ...
- 3 ticks
  - $\kappa = 1$, compaction = 282
  - $\kappa = 2$, compaction = 12
  - $\kappa = 3$, compaction = 0
  - $\kappa = 4$, compaction = 0

Outlook

- Analyze κ for non-trivial programs
- Find optimal middle ground for memory consumption and compaction overhead
- Formalize relation of κ and `refresh()`/`tick()`