The Sticky Mark-Bit Algorithm

Also an intro to mark-sweep GC

7 Oct 2022 – Igalia

Andy Wingo

Automatic Memory Management

"Don't `free`, the system will do it for you"

Eliminate a class of bugs: use-after-free

Relative to bare `malloc`/`free`, qualitative performance improvements

- cheap bump-pointer allocation
- cheap reclamation/recycling
- better locality

Continuum: `bmalloc` / `tcmalloc` grow towards GC

Automatic Memory Management

Two strategies to determine live object graph

- Reference counting
- Tracing

What to do if you trace

- Mark, and then sweep or compact
- Evacuate

Tracing O(n) in live object count

Mark-sweep GC (1/3)

```
freelist := []

allocate():
  if freelist is empty: collect()
  return freelist.pop()

collect():
  mark(get_roots())
  sweep()
  if freelist is empty: abort
```

Mark-sweep GC (2/3)

```
mark():
  worklist := []
  for ref in get_roots():
    if mark_one(ref):
      worklist.add(ref)
  while worklist is not empty:
    for ref in trace(worklist.pop()):
      if mark_one(ref):
        worklist.add(ref)

sweep():
  for ref in heap:
    if marked(ref):
      unmark_one(ref)
    else
      freelist.add(ref)
```

Mark-sweep GC (3/3)

```
marked := 1

get_tag(ref):
  return *(uintptr_t*)ref
set_tag(ref, tag):
  *(uintptr_t*)ref = tag

marked(ref):
  return (get_tag(ref) & 1) == marked
mark_one(ref):
  if marked(ref): return false;
  set_tag(ref, (get_tag(ref) & ~1) | marke
  return true
unmark_one(ref):
  set_tag(ref, (get_tag(ref) ^ 1))
```

Observations

Freelist implementation crucial to allocation speed

Non-contiguous allocation suboptimal for locality

World is stopped during `collect()`: “GC pause”

`mark` O(n) in live data, `sweep` O(n) in total heap size

Touches a lot of memory

Optimization: rotate mark bit

```
flip():
  marked ^= 1

collect():
  flip()
  mark()
  sweep()
  if freelist is empty: abort

unmark_one(ref):
  pass
```

Avoid touching mark bits for live data

Reducing pause time

Parallel tracing: parallelize `mark`. Clear improvement, but speedup depends on object graph shape (e.g. linked lists).

Concurrent tracing: `mark` while your program is running. Tricky, and not always a win ("Retrofitting Parallelism onto OCaml", ICFP 2020).

q*Partial tracing*: `mark` only a subgraph. Divide space into regions, record inter-region links, collect one region only. Overhead to keep track of inter-region edges.

Generational GC

Partial tracing

Two spaces: nursery and oldgen

Allocations in nursery (usually)

Objects can be *promoted*/*tenured* from nursery to oldgen

Minor GC: just trace the nursery

Major GC: trace nursery and oldgen

“Objects tend to die young”

Overhead of old-to-new edges offset by less amortized time spent tracing

Generational GC

Usual implementation: semispace nursery and mark-compact oldgen

Tenuring via evacuation from nursery to oldgen

Excellent locality in nursery

Very cheap allocation (bump-pointer)

But… evacuation requires all incoming edges to an object to be updated to new location

Requires precise enumeration of all edges

JavaScriptCore

No precise stack roots, neither in generated nor C++ code

Compare to V8's `Handle<>` in C++, stack maps in generated code

Stack roots *conservative*: integers that happen to hold addresses of objects treated as object graph edges

(Cheaper implementation strategy, can eliminate some bugs)

JavaScriptCore

Automatic memory management eliminates use-after-free…

…except when combined with manual memory management

Prevent type confusion due to reuse of memory for object of different shape

`addrof/fakeobj` primitives: `phrack.org/issues/70/3.html`

Type-segregated heaps

No evacuation: no generational GC?

Sticky mark bit algorithm

```
collect(is_major=false):
  if is_major: flip()
  mark(is_major)
  sweep()
  if freelist is empty:
    if is_major: abort
    collect(true)

mark(is_major):
  worklist := []
  if not is_major:
    worklist += remembered_set
    remembered_set := []
  ...
```

Sticky mark bit algorithm

Mark bit from previous trace “sticky”: avoid `flip` for minor collections

Consequence: old objects not traced, as they are already marked

Old-to-young edges: the “remembered set”

Write barrier

```
write_field(object, offset, value):
  remember(object)
  object[offset] = value
```

JavaScriptCore

Parallel GC: Multiple collector threads

Concurrent GC: `mark` runs while JS program running; “riptide”; interaction with write barriers

Generational GC: in-place, non-moving GC generational via sticky mark bit algorithm

Alan Demers, “Combining generational and conservative garbage collection: framework and implementations”, POPL ’90

Conclusions

A little-used algorithm

Motivation for JSC: conservative roots

Original motivation: conservative roots; write barrier enforced by OS-level page protections

Revived in “Sticky Immix”

Better than nothing, not quite as good as semi-space nursery

Other considerations

The following slides are just things to think about

Sweeping

Sweeping still O(n): get it out of `collect`

Lazy sweeping: sweep as needed, in `allocate` instead of `collect`; good locality

Concurrent sweeping: sweep in a thread

Allocation

Dynamically switch between freelist and bump-pointer depending on fragmentation

Mitigate freelist overhead by preallocating in thread pools? Manuel Serrano, “Of JavaScript AOT Compilation Performance”, ICFP 2021

Mutator overhead

Representation of remembered set: card table, array, conditional or not, …

Elide write barrier when source object known to be young, e.g. during initialization

Coalesce barriers for multiple writes

Avoid read barriers at all costs

Tracing

How to handle mark stack (`worklist`) overflow

Inline or out-of-line mark bits

Multiple colors for concurrent marking

“Slop”: objects you could have collected if you did a STW full GC, but didn’t

Heuristics: when to pause

Parallel mutators

Not an issue for JavaScriptCore p
Otherwise can be very tricky

Other

Heap iterability

Support conservative roots via is-an-object predicate