From Stack to Heap and Back

Contemporary Currents in Garbage Collection

6 June 2024—Compilers Team Internal Presentation

Andy Wingo

Agenda

Garbage collection in JS implementations over time

The unstable equilibrium of the now

Discussion

But first, a quiz

Allocation algorithms

Collection algorithms

Ways in which different objects are managed differently

Ways to minimize pause times

Difference between concurrent and parallel

Difference between concurrent and incremental

What’s a write barrier for

What’s a read barrier for

Good afternoon

The distant past

The field 20 years ago: IE, Firefox (2004), KHTML

SpiderMonkey: Pile of hoary C++

JS: Gmail (2004), Google Maps (2005)

Low expectations

GC expertise: Java, Lisp, ML

GC on the cheap

SpiderMonkey: Stop-the-world mark-sweep with conservative stack scanning

V8: 2008

Strongtalk / JVM heritage

Baseline compiler with inline caches, hidden classes

GC becomes bottleneck

Generational stop-the-world

- Nursery: Evacuating scavenger
- Old generation: Mark-compact (I think...)

Precise rooting via `Handle<>`

The JS arms race

JSC, SM, V8 engage in race for speed

Compiler work pinches GC

SM: Adopt V8 design; 5-year project to switch to precise roots `https://blog.mozilla.org/javascript/2013/07/18/clawing-our-way-back-to-precision/`

Also, write barrier to tabulate old-to-new edges

2013-2022

Benchmarks measure latency

Push to reduce pause time

Multiplication of cores, rise of mobile: parallelize all the phases, concurrent/ incremental trace

Convergence?

Three-tier runtime (interpreter, baseline, optimizing)

Two-generation GC (scavenger + mark-compact)

Concurrent major trace, lazy/ concurrent sweep, parallel workers

End of history?

Antithesis

JSC uses older GC design: Bartlett Mostly-Copying Collector

- Only some nodes can be target of edges from stack
- The rest can move
- Generational via sticky mark bit `https://wingolog.org/archives/2022/10/22/the-sticky-mark-bit-algorithm`

Why does JSC keep scanning stack conservatively?

CSS: Suxxx?

Cheap to implement

Let the optimizing compiler optimize

- GCC/LLVM can register-allocate temporaries, use internal pointers
- Same for JS optimizing JIT

Still have to pay write barrier cost for on-heap mutations

No overhead for handle management

Risks low: stack often empty, 64-bit address space

Meanwhile, DOM

JS embedded in web browsers

DOM has thousands of object kinds

DOM objects can reference JS

DOM maintained by separate team

SpiderMonkey: Cycle collector

V8: Weak refs from DOM to JS

Bugs happen, they are exploitable

V8: Oilpan

GC provides comprehensive memory safety

Make GC trace C++ object graph

A second GC!

Now in V8: cppgc

V8 GC team starts to own cppgc allocations

Opportunity: Bump-pointer nursery?

Spanner in the works

Many DOM users don’t expect evacuation, e.g. assume that `this` does not change within a method

V8’s scavenger requires users to allow evacuation

Would be nice: fast bump-pointer allocation, but non-moving GC

Synthesis

V8: *Minor mark-sweep nursery*

Instead of evacuating, mark survivors

Block-structured heap, spatially partitioned generations

Promote whole pages instead of individual objects

MinorMS challenges

Hard to beat evacuation / semispace for low survival rates, and survival rates are usually low

- Evacuation work proportional to live size, sweep work proportional to heap size
- Marking needs worklist, evacuation uses simple cheney algorithm
- Compacting has cache benefits
- Evacuation produces lovely bump-pointer arenas

MinorMS opportunities

No need for 2x space

Direct handles instead of indirect

Bump-pointer allocation into regions for cppgc

MinorMS status

On for a % of Chrome users, but not stable

Sticky mark bit experiments: promote by leaving mark bit instead of promoting whole page

Synthesis?

Alice’s Restaurant

Remember Alice? This is a song about Alice

Guile

Boehm-Demers-Weiser single-generation parallel stop-the-world mark-sweep GC with conservative root-finding

Pretty good, actually!

Could be better

Whippet

New GC for Guile. Embed-only libary.

- No-overhead abstract API
- Set of implementations
- A specific Immix-based impl

Whippet impl: Immix-based mark-region collector with compaction via optimistic (fallible) evacuation.

Possibly parallel, generational, conservative stack scanning, conservative heap scanning

`https://github.com/wingo/whippet`

Whippet challenges are like MinorMS challenges

Pinning of stack-referenced objects

Synthesis with object-pinning

Fast allocation

Generation sizing

Sticky mark bit vs block promotion

Work for next 3-4 months or so

Testing Whippet

Whiffle `https://github.com/wingo/whiffle`

AOT baseline compiler for Scheme subset

10 basic microbenchmarks, can run Whippet in all configurations

Whole lot of basic science needed

Up next for Whippet

Heap growth / shrinking

Perfetto / etc

Finalizers

Tuning

Guile integration

Parallel semi-space...

Concurrent marking?

Discussion points

Is V8 making a mistake? SM? JSC?

What are the potential impacts on Node?

What about the sandbox?

What’s next for the dialectic?

What are the impacts of MinorMS on Igalia? Of Whippet?

Commercial ideas