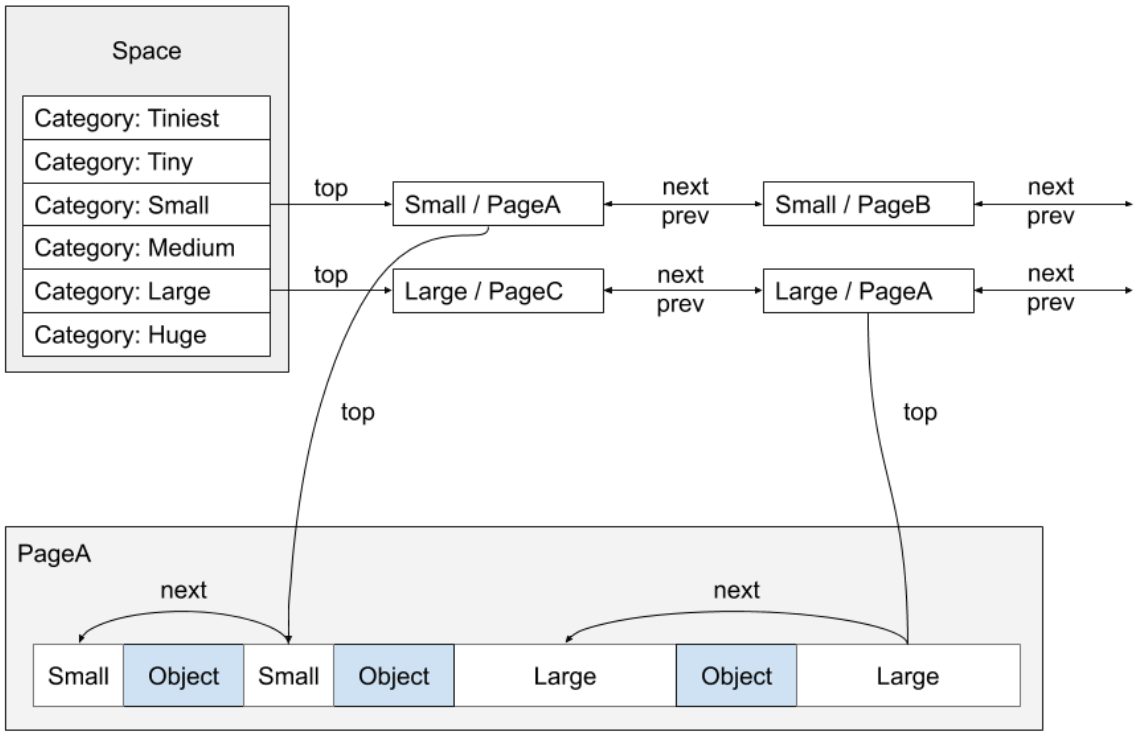
Reducing heap fragmentation in V8

A comparison of various freelist-oriented allocation strategies in V8

Images pool: cf slides:
https://docs.google.com/presentation/d/1ZGd0D7aCKuPyaAjrySHGSTJchtat9twuUhPBirWB674/edit#slide=id.g5fc1128632_2_13

Introduction

Generational GC bla bla bla. Different spaces. We mainly care about old space.
Uses bump-pointer allocation. Refills the linear allocation area through freelists. Refilling linear allocation through freelist is better than just using freelists (explanation: it's always better to leave extra memory in the linear area: if it's too little, it will be returned to the freelist anyway)
Freelists refilled by sweeper.
Freelists structure: steal Ulan's image, but make more generic (or keep it? Since it describe V8's freelist before our work?)



V8 specifics

Different clients of the old space:

- Generated code
- Runtime
- Garbage Collector (scavenger and compactor)
- \+ Add numbers (average or details?) :
    - synthetic: ratio is 10-1-1 (lots of pretenuring).
    - Real-world: ratio is ~0-1-1 (almost no pre-tenuring; depends on the benchs)

Different mechanism to get memory in old space (remove? No because that explain why we don't need lock on freelists/pages) :

- Through the space's freelist
- Stealing pages from the space -> in order to avoid lock

Trade-offs / Design space

General trade-offs / related work

Various freelists:

- A single FreeList (eg, OCaml) -> expensive lookup. Too much memory, to few space. Need to stop scanning after *n* elements in order not to have a very bad worst case.
- Pages segregated by size-class (eg, Go, Spidermonkey)
- Multiple Freelists per page (eg, V8, Dart)
- Small pages vs large pages?
    - Large pages **require** compaction. Extreme case: Speedometer2: with a precise strategy (FreeListMany), old space is x1.5 larger without compaction. Explanation: Probably a burst of allocations at some point, which caused the heap to grow, but a few of those allocations were long-lived and blocked a lot of pages) (even extremer case: I've seen map space being like 750k reserved for < 100k used; don't remember the benchmark though)
    - Large page: cannot segregate page per size-class -> need n freelists per page -> the more freelists, the more overhead (already mentioned in paragraph "Freelist": 100 freelists * 1000 pages = 400k memory overhead)
    - Issues with small pages:
        - More large objects (more mmap, longer linear scanning, queued for unmap)
        - Need GC to scale with more pages
        - Allocation folding (-> not possible to segregate pages by size-class)
            - Turboban's
            - CSA's

-> We focus on V8 -> large pages & multiple freelists (future work: compare with small segregated pages)

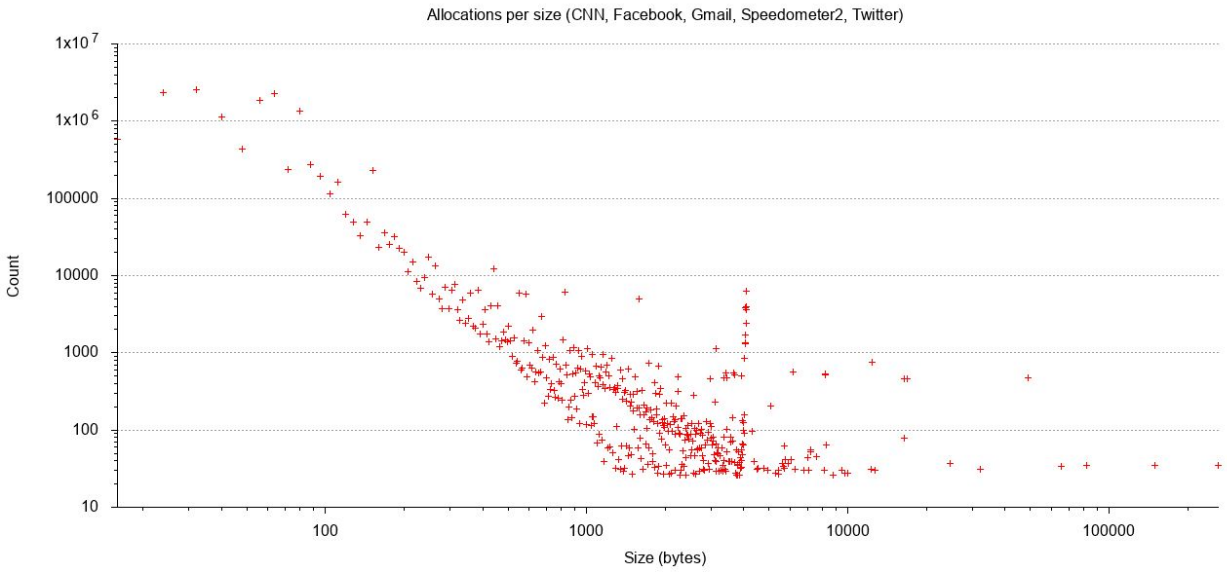
Precise vs Fast

Remove if overlap with introduction?

- Precise (slow)
  - Iterating whole category is not too bad for performances/memory:
    - FreeListLegacySlowPathRetry vs FreeListLegacyMoreSmallsSlowPath:
      - Retry -> -1.2% old space size avg. Explanation: More precise on larger categories
      - Perf-wise: often similar, except on Speedometer2: -5% for Retry (explanation: probably some long pauses in some cases)
  - But can lead to long pause time -> not reasonable in real-time (ie, JS)
  - Mention OCaml: starts scanning at last found.
- Fast.
  - Goal: do more bump-pointer and less freelists allocations.
  - Can be more or less precise still:
    - Linear vs exponential large categories
    - Over-allocate by how much?

Insights

- Most allocations are small -> more important to speed up small allocations (also, any allocation is likely to be followed by small allocations -> overallocating will benefit following few small allocations)



Octane:

- 77% allocs <= 80 bytes
- 93% allocs <= 128 bytes
- 99% allocs <= 256 bytes

Real-world (cnn, discourse, earth, facebook, google_india, maps, nytimes, reddit, twitter, youtube) (same numbers for speedometer2):

- 87% <= 80 bytes
- 93% <= 128 bytes
- 98% <= 256 bytes

- Lots of allocations followed by an allocation of the same size -> over-allocating by 2x can speed up the next allocation. (omit? Because we don't use this in practice)
  - Real-world: 30-40% of allocations followed by allocation of the same size
    - Even more true for smaller size: Up to 50%+ for sizes <= 80 bytes
  - Synthetic: 40-60% of allocations followed by allocation of the same size

  TODO: benchmark on FreeListManyMore? (because no impact on FreeListMany…)
  -> probably won't have an impact as well (really small, hard to measure…)

FreeLists

(TODO: clear distinction between this section and Evaluation)

How we bench:

- Real-world benchmarks
- --predictable-gc-schedule
- Looking at:
  - Old_space effective size
  - JS/C++ duration
  - Scavenger time?

Baseline: FreeListLegacy

6 categories. Boundaries: 24 - 80 - 248 - 2k - 16k - 64k.

- Current strategy: tries to allocate 2k+, then fallback to precise category.
- -> Actually pretty fast (large categories -> large overallocations)

- Without fast path: doesn't impact performances that much. Explanation: boundaries are so imprecise that precise path still overallocates quite a lot.
- (omit?) (Small issue: because cat not precise, a cat can be blocked by smalls elements (eg, if the 24-80 cat starts with 5 times 24; then all allocations > 24 will fail))
- Experiment: discarding tiny and tiniest. Expectation: exact same behavior. Result: Actually uses more memory (~4%). But Scavenger is faster (~1.5%). Explanation: ???

Exploring memory/perf trade-offs

Fast freelists

- FreeListSinglePage: bump-pointer only
  - Too much memory (~ x1.3)
  - Too slow (~ x1.01-1.02). Explanation? Mmap overhead?
  - Doing a bit more compaction make it more reasonable (memory -> x1.05 avg, perf x1 avg). But evacuation time x2-3 -> longer pause time.

- FreeListFast: FreeListLegacy, but discards anything of size < 2k; 3 categories
  - The idea is that small categories don't offer too much overallocation -> this will trigger more expansion, and should use more bump-pointer than FreeListLegacy.
  - Too much memory (~ x1.04 avg)
  - Barely faster, if faster. Explanation: the common case is small allocations, and is already quite optimized with FreeListLegacy.

- FreeListLegacy would come here in the list

Memory-oriented / more versatile freelists

Motivation: FreeListMany: When precise, up to -10% memory, -5% average.
Goal: Finding out the consequence of using such or such category, memory and performance-wise

Note: the more categories, the larger the overhead to store them (1 cat = 4 bytes/page; ie, 1000 pages (250MB memory), 100 cat = 400KB)

Half-region vs full-region

Background: V8's freelist stops at 65k (there is a single freelist for 65k+), even though objects up to 128k can be allocated. Question? Should we have more freelists above 65k.

Experiment: Precise small (<=256 bytes) categories; linear larger categories (1k or 4k grain) (linear larger categories because if exponential larger categories, then since 65k = half the space, there won't be much categories above 65k). Comparing: either stopping larger cats at 65k, or going to 128k.

Results (identical for slow and fast path)

- Full-space: +2% scavenger time (regression)
    - Explanation: More precise large categories = less overallocation.
- Full-space: slight memory improvement, but which is probably countered by overhead of having more freelists

- Real-world benchmarks (cnn, discourse, earth, facebook, google_india, maps, nytimes, reddit, twitter, youtube, Speedometer2): 1629 allocations >= 65k bytes ===> 18 looked past the first element; among which 14 found in the 2nd, 3 in the 3rd, and 1 in the 4th. -> OK to have just one category here.

-> Better map only half the region.

Small categories sizes

Question: how many small (< 256 bytes) categories should we have?
Experiment (+results):

- FreeListMany vs FreeListHalfSmallMany:
    - FreeListHalfSmallMany slightly faster (0% overall; but 0-2% scavenger), but uses slightly more memory (0-1% old heap).
    - 14 less categories in FreeListHalfSmallMany -> 56 less bytes per page (0.02%) -> negligible
- FreeListLegacySlowPath vs FreeListLegacyMoreSmallsSlowPath:
    - FreeListLegacyMoreSmallsSlowPath:
        - -0--1% perf (-0.5% avg)
        - 0-12% less memory (4.4% avg)
        - +3- -8% Scavenger time (-1.8% avg)
    - Explanations: FreeListLegacy overallocates even in the slow path because it's not precise. Furthermore, because it's not precise, small allocation can easily fall into a larger category, thus being overallocated even more.

Fast path overallocation

Question: by how much should be overallocate in the fast-path?

Experiment 1: FreeListManyMore1/2/4k: Precise small (<= 256 bytes), linear large (1k, 2k, 4k); overallocation by 1/2/4k depending on the large categories.

Results:

- 1k -> 2k: +0-1% memory (1k better)
- 1k -> 4k: +0-2% memory (1k better)
- 1k -> 2k: ~-3% scavenger time (2k better)
- 1k -> 4k: ~-5% scavenger time (4k better)
- Performance-wise (JS/C++ duration): nothing too noticeable (wilcoxon not significant)

Explanations:

- (everything is as expected)
- Memory is easy: more overallocation -> less space left for larger objects
- Performance: larger overallocation -> more bump-pointer. Especially in the Scavenger: more over-allocation = need to steal less pages (is this really what's happening?)

Experiment 2: FreeListManyFastPath(/256): Precise smalls (<= 256 bytes), exponential-ish larger (2k-3k-4-6-8-12-16-24-32-48-64k+). Trying to do fast path: +256 bytes vs +2k bytes.

Results:

- +256:
  - 0-1% regression perf (<0.5% avg)
  - 0-3% memory improvement (~1% avg)
  - 0-2% scavenger time regression (~1% avg)

Explanations: as expected.

Linear vs exponential(ish) larger categories

Background: our current strategy uses 3 large categories (2k-16k-65k+). We've evaluated a bunch of linear large categories (1/2/4k) above.
Question: something in between?

Experiment: FreeListLegacy (very exponential (256-2k-16k-65k)) vs FreeListHalfManyFastPath (exponential (2k-4k-8k-16k-32k-65k)) vs FreeListHalfSmallsManyFastPath (exponential, but a bit less (HalfSmalls to be fair with FreeListHalfMany)
(2k-3k-4k-6k-8k-12k-16k-24k-32k-48k-65k)) vs FreeListManyMore2kFastPath (linear, every 2k).

Results:

- Old space size: Legacy < HalfManyFast (-0-4%; avg -1.7%) < HalfSmallManyFast (avg -2.3%) <= ManyFast == ManyMore2kFast (-0-7%; avg -2.8%)
- Perfs: Legacy >= all (-0-1%; avg 0.1%)
- Scavenger:
  - With precise page stealing: Legacy > HalfManyFast (+0-5%; avg 3.5%) > HalfSmallManyFast (+3-5%, avg +4.5%) > ManyFast (+5-8%; avg 6%) > ManyMore2k Fast(+5-10%; avg 7%)

- With larger page stealing: HalfManyFast >= LegacyFast > ManyFast >= ManyMore2kFast

Settling on a FreeList

TODO: Use HalfMany instead of Many (see graph in section evaluation: HalfMany seems quite clearly better: same performances and memory, but faster scavenger)
(and fix numbers below accordingly)

- Different strategies depending on runtime / generated code / GC (scavenger):
  - Runtime / Generate code need to be fast -> overallocation
  - GC (Scavenger) is parallel: can be slower because amortized on the threads -> precise allocation in the Scavenger.

- FreeListMany: 46 categories
  - 30 categories from 24 to 256 bytes (8 bytes precise)
  - 16 categories from 256B to 65k+: exponential-ish
  - Memory-mode (precise):
    - Memory: -0-12%; -5%avg
    - Perf: -0-3%; -1.5% avg
    - Scavenger: -3-9%; -6% avg
  - Performance-mode (overallocation +2k):
    - Memory: -0-7%; -3%avg
    - Perf: -0-1%; 0%avg
    - Scavenger:
  - No compaction:
    - Memory: -3-+3% (except Speedometer: -34%)
    - Perf: same as with compaction
    - V8-gc-evacuate (avg): -6.5%avg
    - V8-gc-evacuate (max): -36%avg

- Fast path:
  - 2k+ overallocation
  - Fallback 256B-2k for small (<=128bytes) objects
    - Impact: benchmarked with 30 retries: within the noise… (remove?)
  - Fallback precise

Reducing memory in non-compacting mode

If pages are “randomly” chosen for allocation, and there is no compaction, space will become fragmented.
This can be partially avoided by allocating in priority in most full pages, and hoping that objects on empty pages die, therefore allowing us to free the pages.
Implemented in our map space: -???% map space size on average

A good way to implement that would be using a Heap (datastructure) rather than a linked-list to store freelists. But not doable in V8 (because linked-list is assumed throughout the GC)

Optimizations

caching empty freelistscategories.

Allows for a better scaling of the number of freelists (only helps performances: memory overhead is still linear in the number of freelists).
Cache comes with O(1) memory overhead; negligible.
Benchmark: FreeListManyMoreFastPath vs FreeListManyMoreFastPathCached (99 categories: linear (+8) 24-256; linear (+256) 256-2048, linear 2048-65k (+1k)).

- C++/JS duration: no significant difference. Explanation: this is already fast without the cache: pre-fetching & co… (and runtime overhead is the same with and without cache)
- Scavenger time: ~-2% (cached is worse)
  Explanation:
  - Cache-misses?
  - The emptier the categories are, the more we overallocate -> the less the cache actually have an impact
  - Slight overhead everywhere to maintain cache

lookup to find category

TODO: Remove this section? Or find some interesting things to say.
not interesting (because small categories computable in O(1), and only a few larger categories compared to the cost of cache misses in the lookup table).

O(1) SelectFreeListCategory

FreeListManyMore (O(1) for all categories) vs FreeListManyMoreLoopSelect (loop for all categories) vs FreeListManyMoreLoopSelectLarge (O(1) for small categories; loop for larger ones). (all precise -> no overallocation).
Result: No measurable difference.
Explanation: More allocs are smalls -> don’t need to loop far.

Sub-conclusion

Don't bother to optimize? This section is a bit weird: we mention optimizations, and for each one we conclude that it's better (or equivalent) without it...

Evaluation

- Synthetic benchmarks. Don't reflect well reality of web: more allocations, more pre-tenuring, not the same allocation patterns.
- Real world benchmarks.

- --predictable-gc-schedule

Looking at:

- Old space max size
- C++/JS duration
- Scavenger time?
- Evacuation time?

Stress-compact vs no-compact:

Octane 2.1:

~20% memory (stress compact better), ~5% perf (no compact better).

Real-world benchmarks:

Stress compact:

- memory: -0-30%; avg -14%
- Perf: -1-2%

No compact:

- Memory: +0-25%; avg 5%
- Perf: <0.3% faster

- the stress compact memory is the lowest we can hope for
- performance-wise, might be possible to do even better than never-compact, by using better freelists?

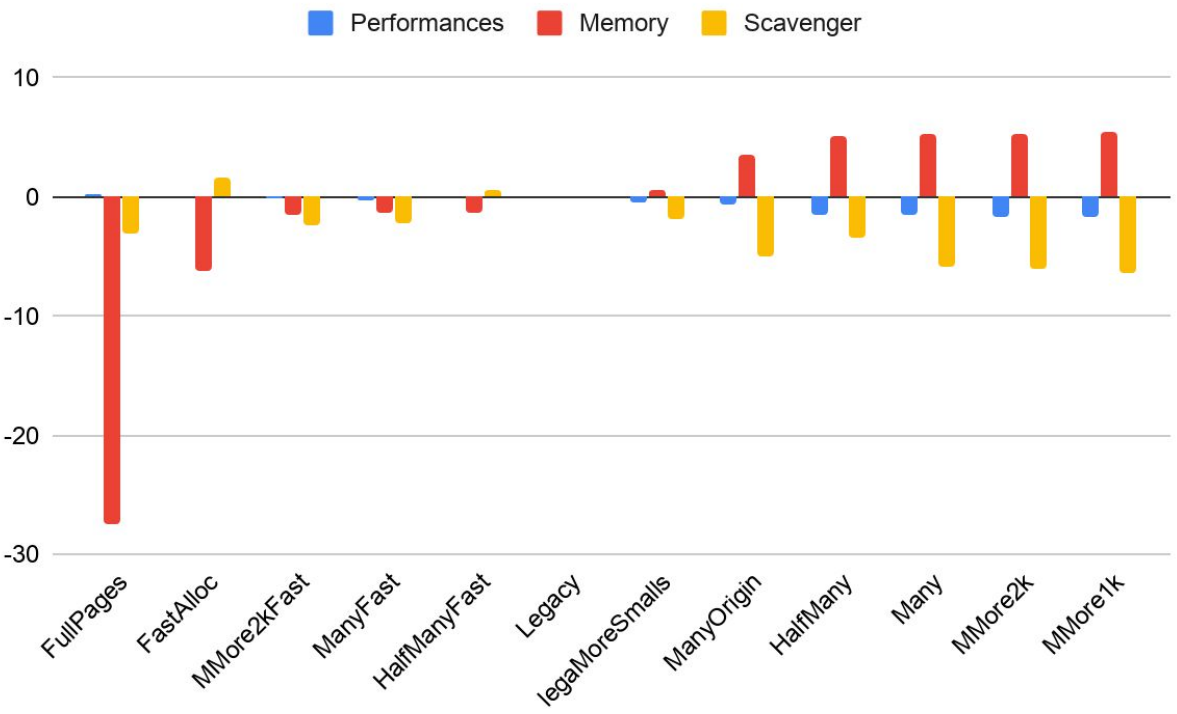
- Never-compact: almost viable with FreeListMany (-> means less pause time in major GC; less write-barrier overhead)

- --reduce-memory (using memory critical heuristic for selecting evacuation candidates) -> improve memory quite a lot but longer pause time.

Comparison of most interesting Freelists

(ordered from (supposedly) better perf to (supposedly) better memory)

```
 "--gc-freelist-strategy=6",          # FreeListFullPages
  "--gc-freelist-strategy=1",          # FreeListFastAlloc
  "--gc-freelist-strategy=0",          # FreeListLegacy
  "--gc-freelist-strategy=7",          # FreeListLegacyMoreSmalls
  "--gc-freelist-strategy=39",          # FreeListHalfManyFastPath
  "--gc-freelist-strategy=38",          # FreeListManyFastPath
  "--gc-freelist-strategy=41",          # FreeListManyMore2kFastPathNoCache
  "--gc-freelist-strategy=40",          # FreeListManyOrigin
  "--gc-freelist-strategy=32",          # FreeListHalfMany
  "--gc-freelist-strategy=2",          # FreeListMany
  "--gc-freelist-strategy=27",          # FreeListManyMore2k
  "--gc-freelist-strategy=10",          # FreeListManyMore(1k)
```



Related Work

(remove if too much overlap with Section Trade-offs & co)

Strategies adopted by other GCs:

- No compaction. Requires small pages. Example: Go.
- Compacting, freelist oriented. Example: Dart (128 freelists).
- Compacting, compaction oriented. Example: OCaml (1 freelist) *(Note: I'm not sure how much OCaml's people would consider their GC as "compaction oriented though").*
- Segregation of pages by size class
    - No linear dependency on the number of freelists
    - No overhead to store freelists

Conclusion

- No silver bullet.
- Use different freelists based on context (optimize for perf vs optimize memory); switch freelist at runtime.

Future work ideas

- Comparison with size-class segregation
- Tune the strategy depending on live feedback (eg, lots of allocations -> bump pointer; few allocations -> precise)
- Evaluate "end of cycle allocation latency": after-GC allocation use fast path are are fast, but after a while, larger categories are empty, and we fall back to the slow path more often. Combined with other effects, it could result in jank toward the end of cycles.
    - Fix by not using fast path for every allocation? (but a bit tricky: fast path for allocation of 24 bytes => next 1k alloc will be fast. Means that just alternating fast path/slow path for refilling freelist will have no impact (since 1k allocs will be fast, and only one will be slow)
    - Make GC scheduler aware of that -> start gc earlier