Temeraire: Hugepage-Aware Allocator

Andrew Hunter, Chris Kennelly

Notes on the name[^cutie]*: the french word for "reckless" or "rash"* 😃*, and also the name of several large and powerful English warships. So: giant and powerful, but maybe a little dangerous.* 😃

This is a description of the design of the Hugepage-Aware Allocator. We have also published "Beyond malloc efficiency to fleet efficiency: a hugepage-aware memory allocator" at OSDI 2021, which provides further details on the design, implementation, and rollout of Temeraire.

GOALS

What do we want out of this redesign?

- Dramatic reduction in pageheap size. The pageheap in TCMalloc holds substantial amounts of memory *after* its attempts to `MADV_DONTNEED` memory back to the OS, due to internal fragmentation. We can recover a useful fraction of this. In optimal cases, we see savings of over 90%. We do not expect to achieve this generally, but a variety of synthetic loads suggest 50% of pageheap is a reasonable target savings.

- Dramatic increase in hugepage usage. The `madvise()` in `ReleaseMemoryToSystem` is made without any thought to transparent hugepages, and in practice prevent most fleet RAM from remaining as intact hugepages. Services have seen substantial performance gains from **from disabling release** (and going to various other lengths to maximize hugepage usage).

- *reasonable* allocation speed. This is really stating a non-goal: speed parity with `PageHeap::New`. PageHeap is a relatively light consumer of cycles. We are willing to accept a speed hit in actual page allocation in exchange for better hugepage usage and space overhead. This is not free but we think is well justified. Our goal is more to avoid catastrophic regressions in speed. We intentionally accept two particular time hits:

    - much more aggressive releasing (of entire hugepages), leading to increased costs for *backing* memory.
    - much more detailed (and expensive) choices of where to fulfill a particular request.

DESIGN

The algorithm -- as usual here, really, the data structures, which neatly determine our algorithm -- are nicely divided into components. Essentially, the path of an allocation goes like this:

1. If it is sufficiently small and we have the space we take an existing, backed, partially empty hugepage and fit our allocation within it.
2. If it is too large to fit in a single hugepage, but too small to simply round up to an integral number of hugepages, we best-fit it into one of several larger slabs (whose allocations can cross hugepage boundaries). We will back hugepages as needed for the allocation.
3. Sufficiently large allocations are rounded up to the nearest hugepage; the extra space may be used for smaller allocations.

Deallocation simply determines which of 1), 2), or 3) happened, and marks the corresponding object we allocated from as free.

We will sketch the purpose and approach of each important part. Note that we have fairly detailed unit tests for each of these; one consequence on the implementations is that most components are templated on the `tcmalloc::SystemRelease` functions[^templated] as we make a strong attempt to be zero initializable where possible (sadly not everywhere).

`RangeTracker`

`RangeTracker` and `Bitmap`, its underlying implementation, are helper class used throughout the components below. They are both quite simple: `Bitmap` is a fixed-size (templated) bitmap with fast operations to set and clear bits and ranges of bits, with extensive support for searching and iterating. (Search and iteration support is why `std::bitset` is not usable here.)

`RangeTracker` is essentially a `Bitmap` augmented with statistics on usage, in particular the longest range of contiguous free (false) bits. It provides methods to do best-fit allocation from free ranges (keeping the statistics correct).

Both of these need to be quite fast as they're on nearly every allocation/deallocation path in `HugePageAwareAllocator` (in multiple ways)! They are reasonably optimized but probably still have more headroom.

HugeAllocator/HugeCache (the backing...)

This is a set of classes that fulfills requests for backed (or unbacked) aligned hugepage ranges. We use this for sufficiently large (or nicely sized) requests, and to provide memory for the other components to break up into smaller chunks.

`HugeAllocator`

`HugeAllocator` is (nearly) trivial: it requests arbitrarily large hugepage-sized chunks from `SysAllocator`, keeps them unbacked, and tracks the available (unbacked) regions. Note that we do not need to be perfectly space efficient here: we only pay virtual memory and metadata, since *none* of the contents are backed. (We do make our best efforts to be relatively frugal, however, since there's no need to inflate VSS by large factors.) Nor do we have to be particularly fast; this is well off any hot path, and we're going to incur non-trivial backing costs as soon as we're done assigning a range.

The one tricky bit here is that we have to write some fiddly data structures by hand. We would have liked to implement this by grabbing large (gigabyte+) ranges from SysAllocator and using bitmaps or the like within them; however, too many tests have brittle reliance on details of `SysAllocator` that break if TCMalloc consistently requests (any considerable amount) more than the minimum needed to back current usage. So instead we need to track relatively small ranges. We've implemented a balanced tree that merges adjacent ranges; it is, as we said, fiddly, but reasonably efficient and not stunningly complicated.

`HugeCache`

This is a very simple wrapper on top of HugeAllocator. It's only purpose is to store some number of backed *single* hugepage ranges as a hot cache (in case we rapidly allocate and deallocate a 2 MiB chunk).

It is not clear whether the cache is necessary, but we have it and it's not costing us much in complexity, and will help significantly in some potential antagonistic scenarios, so we favor keeping it.

It currently attempts to estimate the optimal cache size based on past behavior. This may not really be needed, but it's a very minor feature to keep *or* drop.

`HugePageFiller` (the core...)

`HugePageFiller` takes small requests (less than a hugepage) and attempts to pack them efficiently into hugepages. The vast majority of binaries use almost entirely small allocations[^conditional], so this is the dominant consumer of space and the most important component.

Our goal here is to make our live allocations fit within the smallest set of hugepages possible, so that we can afford to keep all used hugepages fully backed (and aggressively free empty ones).

The key challenge is avoiding fragmentation of free space within a hugepage: requests for 1 page are (usually) the most common, but 4, 8, or even 50+ page requests aren't unheard of. Many 1-page free regions won't be useful here, and we'll have to request enormous numbers of new hugepages for anything large.

Our solution is to build a heap-ordered data structure on *fragmentation*, not total amount free, in each hugepage. We use the **longest free range** (the biggest allocation a hugepage can fulfill!) as a measurement of fragmentation. In other words: if a hugepage has a free range of length 8, we *never* allocate from it for a smaller request (unless all hugepages available have equally long ranges). This carefully husbands long ranges for the requests that need them, and allows them to grow (as neighboring allocations are freed).

Inside each equal-longest-free-range group, we order our heap by the **number of allocations** (chunked logarithmically). This helps favor allocating from fuller hugepages (of equal fragmented status). Number of allocations handily outperforms the total number of allocated pages here; our hypothesis is that since allocations of any size are equally likely[^radioactive] to become free at any given time, and we need all allocations on a hugepage to become free to make the hugepage empty, we're better off hoping for 1 10-page allocation to become free (with some probability P) than 5 1-page allocations (with probability P^5).

The `HugePageFiller` contains support for releasing parts of mostly-empty hugepages as a last resort.

The actual implementation uses a fixed set of lists and a bitmap for acceleration.

`HugeRegion` (big but not enormous...)

`HugeAllocator` covers very large requests and `HugePageFiller` tiny ones; what about the middle? In particular, requests that cannot fit into a hugepage, but should not be rounded to multiples? (For instance, 2.1 MiB.) These are woefully common.

In any case, we certainly have to do something with "2.1 MiB"-type allocations, and rounding them to 4 will produce unacceptable slack (see below for what we can do with the filler here; it is wildly insufficient in current binaries which have the majority of their allocation in these large chunks.)

The solution is a much larger "region" that best-fits these chunks into a large range of hugepages (i.e. allows them to cross a hugepage boundary). We keep a set of these regions, and allocate from the most fragmented one (much as with Filler above)! The main difference is that these regions are kept **un-backed** by default (whereas the Filler deals almost entirely with backed hugepages). We back hugepages on demand when they are used by a request hitting the region (and aggressively _unback _them when they become empty again).

A few important details:

- These regions are currently 1 GiB, which is very large!

  The reason is this: suppose our entire binary allocates a huge number `N` of requests of size `S` that are too big for the filler, but that don't evenly divide the region size `M` (say, 2.1 MiB 😃) How much space will we waste? Answer: we will allocate about `R = N / (M / S)` regions, with each region storing `floor(M/S)` allocations. The tail will be unused. We can unback any totally untouched hugepages, but suppose that `M/S` allocations just barely touches the last hugepage in the region: we will then waste ~a full hugepage per region, and thus waste `R` hugepages. Conclusion: the larger a region we use, the less waste (in this case). Originally regions were 32 MiB, and this effect was very noticeable. This also allows us to use very few regions in a given binary, which means we can be less careful about how we organize the set of regions.

- We don't make *any* attempt, when allocating from a given region, to find an already-backed but unused range. Nor do we prefer regions that have such ranges.

  This is basically a question of effort. We'd like to do this, but we don't see any way to do it without making the data structure more complicated and cumbersome. So far in tests it hasn't proved a major problem. (Note that `RangeTracker` has a low-address bias, which will help somewhat here by compacting allocations towards the low end of any region).

Additional details on the design goals/tradeoffs are in the Regions Are Not Optional design doc.

`HugePageAwareAllocator` (putting it all together...)

This class houses the above components and routes between them, in addition to interfacing with the rest of TCMalloc (the above classes don't need or use Spans, for instance). This is mostly straightforward; two points are worth discussing.

- How do we choose which sub-allocator for a given request?

  We use a size-based policy.

  1. Small allocations are handed directly to the filler; we add hugepages to the filler as needed.
  2. For slightly larger allocations (still under a full hugepage), we *try* the filler, but don't grow it if there's not currently space. Instead, we look in the regions for free space. If neither the regions or the filler has space, we prefer growing the filler (since it comes in smaller chunks!) The reasoning here is that if our binary only has allocations of (say) ¾ a hugepage, we don't want the filler to be giant but ¼ empty; but in a more reasonable binary where we can easily pack such allocations near smaller ones, we'd prefer to do so over using the region.
  3. Allocations that won't fit in a hugepage are just given to the regions (or, for truly enormous ones, to `HugeAllocator` directly).

The changeover point between 1) and 2) is just a tuning decision (any choice would produce a usable binary). Half a hugepage was picked arbitrarily; this seems to work well.

- How do we handle backing?

Allocations from `HugeAllocator` or `HugeRegion` (some of the time) need to be backed; so do hugepages that grow the `HugePageFiller`. This isn't free. Page heap allocation isn't hugely expensive in practice, but it is

under a lock and contention matters. We currently rely on access by the application to back memory, and assume returned memory has been backed.

For accounting purposes, we do a bit of tracking whether a given allocation is being fulfilled from previously-unbacked memory.

We do wire that information to the point we drop the pageheap lock; we then back it without producing lock contention. This made a noticeable performance difference when explicitly backing memory before returning it to the application.

Notes

[^cutie]: Also the name of this cutie, the real reason for the choice. [^templated]: It will be possible, given recent improvements in constexpr usage, to eliminate this in followups. [^conditional]: Here we mean "requests to the pageheap as filtered through sampling, the central cache, etc" [^radioactive]: Well, no, this is false in our empirical data, but to first order.