



▲ [microsoft/mimalloc](#) [programming](#) [github.com](#)  
 41  via wizardishungry 5 months ago | [archive](#) | 13 comments

▲  calvin 5 months ago | [link](#)

9 This is probably bait for David, but how does this compare to other allocators, like snmalloc?

-] ▲  david\_chisnall 5 months ago | [link](#)

24 They are quite similar in a lot of ways. Both allocate memory in chunks, assign chunks to size classes, and then do allocation by mapping sizes to size classes and allocating an object from a chunk assigned to that size class. Both use a lock-free message queue to return memory to the source.

The big difference between the two is that mimalloc has a message queue per chunk and returns memory directly into the free list, snmalloc has a message queue per allocator (typically, one per thread) and batches messages. The direct approach in mimalloc means that they need a compare-and-swap on their free path. Snmalloc needs only an exchange to enqueue messages. Snmalloc also batches messages, so needs an atomic operation only when a thread has allocated a certain number of foreign objects (I believe this is 1 MiB by default, it's configurable). In practice, because mimalloc has so many free lists, the probability of contention on their CAS is very low. Snmalloc uses a temporal radix tree for messages. Each allocator has a unique ID (typically the address of its message queue, though this is also configurable). Outbound messages are assembled into a smallish number of lists using the low bits of the ID and then sent to whichever allocator is the owner for the item on the front of the list. When you receive messages, if they're not aimed directly at you then you put them into the list for the next few bits, and so on. This means that sometimes memory takes two hops (or more in systems with very large thread counts) to get home but means that the assignment of messages to queues is very fast.

We stole^Wadapted a lot of great ideas from mimalloc. We used to have a separate bump allocator and free list. Now, when the free list in a slab is empty we hit a slow path and turn a complete page of memory into a free list. My favourite trick from mimalloc is that we initialise our thread-local allocator pointer to point to a global allocator that doesn't own any slabs. We then check before we allocate any slabs if we're the global allocator and, if so, allocate an allocator. This means that we can move the 'is the thread-local allocator' check completely off the fast path, so we're on something like 12 x86 instructions for the fast path.

I think we are now faster than mimalloc in their benchmarks, though there's a lot of cross-pollination of ideas so don't expect this to remain the case forever. There are a few things that, in my incredibly biased opinion, make snmalloc better:

- It's a C++ codebase, so platform and architecture abstraction layers are clearly separated in PAL and AAL classes that make porting it trivial.
- We use C++ template pointer wrappers for tracking the difference between pointers to objects on a free list, pointers to objects that have been handed out to the client, pointers to objects within a slab that can be used to index anywhere into the slab, and so on. This makes security auditing easier and also makes it easier for us to support things like ChERI and MTE. Robert Norton-Wright on my team is working on a combination ChERI+MTE design using snmalloc.
- We separate a lot of policy and mechanism. There's a Backend class that defines how you assemble the various building blocks in snmalloc, making it easy to use in different situations.
- The single message queue per allocator means that we have a single interception point for when some of the allocators in the system are untrusted. We're using this with the Verona unsafe code sandboxing so that we can do cheap allocation inside and outside the sandbox and just have to

verify that pointers that the allocator on the outside receives remain in the sandbox. This gives us a really nice primitive for building an RPC layer: in both directions, it's very fast to allocate memory that the other side can free.

Friendly competition from mimalloc has been great for improving snmalloc.

I can't remember if mimalloc does this too, but we have a very cheap way of finding the start and end of an allocation. We can use this to do bounds checks of heap libc functions. For example, we can add bounds checks to memcpy, with a fairly small overhead. I plan on trying to upstream that to FreeBSD soon - on both x86 and PowerPC our C++ memcpy either outperforms the hand-written assembly in FreeBSD libc or is the same speed within the range of experimental error. The same is true for glibc memcpy. Musl's x86 memcpy is embarrassingly bad: they've written something in assembly that's slower than a naive C version - it's 2-10x slower than ours for most sizes under 256 bytes, even with the bounds checks.

[ - ] ▲ 🌸 david\_chisnall 5 months ago | link

5

By the way, snmalloc builds a .so out of the box that you can LD\_PRELOAD can replace malloc and friends. If anyone is willing to try it, we'd love to hear real-world performance data. I think the things Matt wanted to do to fix the performance regressions from the big snmalloc 2 refactoring are done now. I'd also be especially curious about the performance of the memcpy mitigation. Our friends in MSRC tell us that around 10% of arbitrary code execution vulnerabilities start from someone getting the bounds to memcpy wrong, so if that's fast enough for folks to be able to enable in production then it's a big win for security.

[ - ] ▲ 🌸 Moonchild 5 months ago | link

2

*our C++ memcpy either outperforms the hand-written assembly in FreeBSD libc or is the same speed within the range of experimental error. The same is true for glibc memcpy*

That's no mastery! :P FreeBSD strings functions are good, but not vectorised; and glibc is ok, but not great.

[ - ] ▲ 🌸 david\_chisnall 5 months ago | link

5

The thing that surprised me is that all of these implementations are hand-written assembly. Mine is high-level C++. On x86, I have one instruction of inline assembly for using `rep movsb` for very large copies. The architecture-specific parts are isolated in a separate class that defines the largest width to use for a single copy and allows tuning of how you copy different sizes but even with the default version the PowerPC64 implementation was very close to the performance of the hand-written assembly in glibc. The tuned version *with* heap bounds checks is faster than the glibc assembly version until you get to large sizes.

Writing assembly is fun but if you're doing it for performance then you need to be really sure that the compiler is the problem. As far as I can tell, for memcpy, it isn't.

[ - ] ▲ 🌸 Moonchild 5 months ago | link

2

You want to be able to tune the branch order/direction for real workloads. The compiler has its own ideas about what order branches should go in (as well as otherwise messing with control flow, changing which stores go where, etc.); likely/unlikely is unlikely to cut it, and pgo lies as it does not account for the cost of actually performing the copy. For this sort of performance work, I also think assembly is *easier*, as it is more transparent. You may be able to coax the compiler into generating good-enough code; but if you are staring at disassembly, you already have your own ideas about what that disassembly should look like, and wouldn't it be easier to just write it yourself?

[-] ▲ 🌟 david\_chisnall 5 months ago | link

3

The small cases in mine are generated by some recursive templates that are specialised based on the target's register size (so they expand to a single computed jump and then descending power-of-two loads and stores exactly matching the size). I probably *could* write these in assembly for a single architecture and it would only be, maybe, twice as hard as doing it in C++. Doing it for half a dozen architectures would be much harder. The end result is sufficiently complex that I wouldn't be confident that I could write something as fast, let alone faster, in assembly.

Oh, and with the templated C++ version it's trivial for me to instantiate it multiple times for different register widths to generate `ifuncs` based on the available hardware that use different vector register widths, for example. I think the real problem with the assembly version is the agility. With the C++ version, I tried a few dozen very different variations, constructed from the same templated building blocks with different parameters, in an hour. Doing the same thing with hand-written assembly would be much harder. The optimal memcopy for a given architecture changes over time as microarchitectures (and available instructions) change and flexibility is a big win. Using AVX instead of SSE for mine requires changing one constant and adding one compiler flag (I didn't do this because it ended up being slower on the systems I tried but in the future it might make sense). Even doing that experiment would be a complete rewrite with assembly, unless you used a *lot* of macros (and even then, only if register allocation is easy).

▲ 🎓 Student 5 months ago | link

2 Well that's quite exciting. Anyone used it with go lang?

[-] ▲ ⌚ 4ad [Go Contributor](#) edited 5 months ago | link

3

Go uses its own memory allocator. In fact, Go doesn't use any C library at all (except sometimes `libc`, but even then, it only uses `libc` to call system calls and things like `getpwuid_r`, not for `malloc`).

[-] ▲ 🧙 wizardishungry 5 months ago | link

3

I think they may have been thinking of something like this: [Manual Memory Management in Go using jemalloc](#)

[-] ▲ 🎓 Student 5 months ago | link

1

Thanks for that. Tbh I was hoping to get "free" performance improvements because I know the codebase I work on allocates quite freely.

[-] ▲ 🎓 Student 5 months ago | link

2

Right you are. <https://github.com/golang/go/blob/master/src/runtime/malloc.go>

▲ 🧑 skade 5 months ago | link

2

Fun bit: `mimalloc` uses ASLR for randomness as a weak fallback if secure randomness isn't available. <https://github.com/microsoft/mimalloc/blob/15220c684331d1c486550d7a6b1736e0a1773816/src/random.c#L255>