snmalloc: A Message Passing Allocator

Paul Liétar
Microsoft Research, UK
paul@lietar.net

Theodore Butler*
Drexel University, USA
theodore.j.butler@drexel.edu

Sylvan Clebsch
Microsoft Research, UK
Sylvan.Clebsch@microsoft.com

Sophia Drossopoulou
Imperial College London, UK
s.drossopoulou@imperial.ac.uk

Juliana Franco
Microsoft Research, UK
Juliana.Franco@microsoft.com

Matthew J. Parkinson
Microsoft Research, UK
mattpark@microsoft.com

Alex Shamis
Microsoft Research, UK
Imperial College London, UK
alexsha@microsoft.com

Christoph M. Wintersteiger
Microsoft Research, UK
cwinter@microsoft.com

David Chisnall
Microsoft Research, UK
David.Chisnall@microsoft.com

Abstract

Snmalloc is an implementation of malloc aimed at workloads in which objects are typically deallocated by a different thread than the one that had allocated them. We use the term *producer/consumer* for such workloads. Snmalloc uses a novel message passing scheme which returns deallocated objects to the originating allocator in batches without taking any locks. It also uses a novel *bump pointer-free list* data structure with which just 64-bits of meta-data are sufficient for each 64 KiB slab. On such producer/consumer benchmarks our approach performs better than existing allocators.

Snmalloc is available at https://github.com/Microsoft/snmalloc.

CCS Concepts • **Software and its engineering → Allocation / deallocation strategies**.

Keywords Memory allocation, message passing

ACM Reference Format:
Paul Liétar, Theodore Butler, Sylvan Clebsch, Sophia Drossopoulou, Juliana Franco, Matthew J. Parkinson, Alex Shamis, Christoph M. Wintersteiger, and David Chisnall. 2019. snmalloc: A Message Passing Allocator. In *Proceedings of the 2019 ACM SIGPLAN International Symposium on Memory Management (ISMM '19), June 23, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3315573.3329980

[*] Work done while at Microsoft Research.

ISMM '19, June 23, 2019, Phoenix, AZ, USA

ACM ISBN 978-1-4503-6722-6/19/06…$15.00
https://doi.org/10.1145/3315573.3329980

1 Introduction

Individual threads typically cache free objects to improve performance of allocators in multi-threaded scenarios: rather than returning deallocated objects to the original allocating heap/slab, threads keep deallocated objects in size-specific thread-local lists and so can quickly reallocate objects of that size when needed. In fact, almost all modern allocators employ some form of thread-caching [8, 10, 13].

When allocations and deallocations are roughly evenly spread amongst threads, these caches do not grow too large and no synchronization across threads is necessary. This leads to a dramatic improvement in performance, at the cost of not coalescing free memory that is kept in thread caches.

However, simple thread-caching does not work well in all scenarios. Producer/consumer workloads form the most common such scenario, where some threads primarily perform allocation and other threads primarily perform deallocation. If the overall allocation pattern is roughly symmetric, that is, if threads allocate and deallocate roughly the same number of objects of each size, then thread-caching still performs well. If the allocation pattern is asymmetric, that is, if the number of allocations and deallocations of a particular size varies significantly across threads, then thread-caching does not perform well: allocating threads are constantly exhausting their local caches, while deallocating threads are constantly over-filling them. This leads to synchronization, and thus can be expensive. Such asymmetric scenarios occur in pipelined programs, which pass heap-allocated structures between threads. They also occur in some GC implementations: GC threads perform all the deallocations while mutator threads perform all the allocations.

This paper presents snmalloc, a new point in the allocator/deallocator design space. Instead of thread-caching, we use lightweight lock-free message-passing to send batches of deallocations to the originating thread.

While using batched message-passing instead of thread-caching is the main novelty of snmalloc, we also had to address the following design questions:

Simple Communication Allocators communicate upon deallocation only, and communication takes place through message passing. Thus, adopting the asynchronous paradigm leads to a simple design that fits the producer/consumer scenario well. Each allocator can be associated to a single thread and does not need to lock its internal data-structures.

Efficient Message Passing We have adapted a scheme for efficient message passing queues which supports multiple producers and a single consumer. At the cost of loss of linearizability [12] (not essential for queues of deallocation requests), dequeueing requires no synchronization, while enqueueing requires one fence and one atomic exchange.

Effective Dispatching Allocators may send messages to any other allocator. In a naïve implementation each allocator would keep a queue of batched messages for each other allocator. The number of queues would then either be the dynamically known number of existing threads, or the statically known maximal number of possible threads. The former would require allocation of a dynamically sized structure when handling remote deallocation and slow enqueueing, while the latter would lead to significant wasted space and hard-coded limits.

Instead, we adapted ideas from radix trees. Allocators keep a fixed 2^k size array of buckets of pending messages, where k in our implementation is 6. Batched messages are inserted into the bucket which corresponds to their destination's address, modulo the number of buckets. Dispatch of messages takes place by sending all the messages from the same bucket to *one* allocator, which then responds to those messages for which it is the final destination, and forwards the rest, again according to their destination allocator address, but now shifted right by k bits. Thus we gain an efficient data structure that requires no allocation at the cost of intermediate hops upon message send. The number of intermediate hops is bound by $\lceil N/k \rceil$ where N is the number of meaningful bits used by allocator addresses. In our implementation the value of this bound is 7, as allocator addresses are 2KiB aligned in a 48 bit address space, yielding 37 meaningful allocator address bits.

Efficient Discovery of Free Memory Allocators need to be able to find free memory of a certain size. Bit masks are a common approach for maintaining meta-data as to whether or not a part of memory is free. With this representation, and a minimum object size of 16 bytes, we would need 1 bit per 16 bytes of space; hence representing the allocation status of 64KiB of memory would require 512 bytes of meta-data.

Instead, in `snmalloc`, we use a data structure that combines bump pointer style allocation with a free list. Effectively, we have a standard free list, but instead of terminating this list with null, we terminate it with the high-water mark of allocation for this slab of memory. This representation allows us to use just 64 bits of meta-data for each 64KiB slab. It also is very cheap to initialise: setting the head of the list to be the bump pointer to the start of the slab is sufficient.

As is customary, we organize objects of the same size into slabs. We keep the free lists within the free memory of the slabs themselves.

We evaluated `snmalloc` using micro-benchmarks as well as real programs. We compared it with state-of-the-art memory allocators from both industry and research: `Hoard` [2], `jemalloc` [8, 9], `lockfree` [18], `lockless` [13], and also `ptmalloc2` [11], `scalloc` [1], `rpmalloc` [14], `tbbmalloc` [15], `tcmalloc` [10], and `SuperMalloc` [16],

We measured throughput and memory usage using microbenchmarks from the SuperMalloc repository and some of our own. We have also compared using SPEC 2017, and FaRM.

Moreover, because `snmalloc` introduces several new techniques, we expose a large parameter space. We benchmark several points in this space to demonstrate the impact of various design decisions. So far, our evaluation has produced very encouraging results.

2 Implementation

2.1 Large, Medium, and Small Objects, and Allocators

`Snmalloc` is concerned with the allocation and de-allocation of *objects*. We use the term object to describe a consecutive piece of memory that has been allocated as the result of a single allocation, which in turn must be freed as a single unit. Note that, because of representation and alignment concerns, the amount of memory allocated may exceed the amount requested in the `malloc` call — we will discuss this later on.

We distinguish between *large*, *medium* and *small* objects. Large objects require at least 16MiB (2^{24} bytes), medium objects require less than 16MiB and at least 64KiB (2^{16} bytes), and small objects require less than 64KiB. These numbers are configurable and in Section 3 on page 8 we will show the effects of using smaller sizes for each category.

Allocators are responsible for object allocation and deallocation. There is one allocator per thread. Small and medium objects are *owned* by the allocator that allocated them, which is also responsible for their deallocation. Large objects are deallocated centrally using a per-size lock-free stack.

We employ a *page map* and some meta-data which, for any given internal pointer, determines the size of the object and the owning allocator. We will discuss these in Sections 2.3, 2.4, 2.5, 2.7, and 2.8.

In the next section we discuss how an allocator sends batched requests to other allocators when it wishes to deallocate small or medium objects it does not own.

2.2 Message passing allocator

When an allocator begins deallocating an object it uses the page map detailed in Section 2.4 on page 5 to determine whether the object being deallocated is a small, medium, or large object. If the object is a large object, deallocation is handled as described in Section 2.5.

If the object is medium or small, the allocator uses the meta-data of the containing medium slab or superslab, as detailed in Sections 2.7 and 2.8, to identify that object's owning allocator. If the allocator is the owning allocator then it directly modifies the corresponding meta-data of the containing slab to indicate that the object has been deallocated, as detailed in Sections 2.7 and 2.8. Otherwise, the allocator issues a remote deallocation request and transmits it to the owning allocator. These remote deallocation requests are batched for performance. The rest of this section describes how we send batches of requests in an efficient manner.

Each allocator exposes a lock-free multi-producer, single-consumer queue onto which other allocators push these deallocation requests. This design allows any thread to asynchronously dispatch objects that are pending deallocation to their owning allocator at any time. On every call to `malloc` or `free`, the allocator will check its own incoming queue and process any requests it might contain. A batch of messages can be pushed with a single atomic pointer exchange, i.e., not a compare-and-swap and without a loop. Reading from the queue requires no atomic operation at all. On CPUs with weak memory ordering, such as ARM, a release fence on push and an acquire fence on read are required. The implementation of the queue is detailed later in this section.

Each message queue entry contains a pointer to the next object and the target allocator's identifier. We store these inside the deallocated object, avoiding the need to allocate new memory. The smallest size of object is therefore twice the size of a pointer, which is 16 bytes on a 64 bit architecture.

Pushing each request individually onto the target allocator's queue would be inefficient, as it would require one atomic operation per freed object. Instead, outgoing requests are sent in batches. Inside the allocator sending requests, these are grouped into separate linked-lists, based on the destination allocator. Whenever the total size of objects stored in the outgoing lists reaches a configurable threshold (currently 1MiB), the allocator posts the linked-lists to the target allocators' incoming queue. This requires a single atomic operation per destination, regardless of the number of objects.

Temporal Radix Tree Using a separate outgoing list for each destination would either require an upper bound on the number of allocators used by the application, or dynamic allocation of outgoing lists. The former is impractical for a general purpose `malloc` implementation, and the latter would need a separate allocation scheme, which would introduce unwanted synchronization.

Instead, outgoing requests are grouped into *buckets*. This grouping is not by target allocator but by the lower few bits of the target allocator's address. This reduces the number of outgoing lists to 2^k, where k is the number of bits used. In the default configuration, we use $k = 6$ bits, requiring 64 buckets per allocator to store outgoing pending requests.

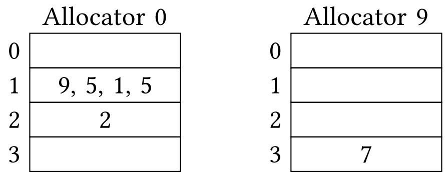
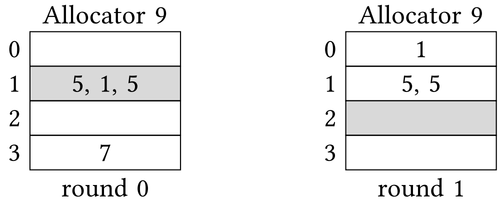


Figure 1. k=2, thus four buckets per allocator. Allocator 0 has pending requests for allocators 1, 2, 5 and 9, and allocator 9 has a pending request for allocator 7.



(a) State of allocator 9 after allocator 0 flushed all its pending requests; home bucket in grey.

(b) State of allocator 9 after its first round of flushing its pending requests; home bucket has changed.

Figure 2. Temporal radix tree for remote deallocation through two rounds

Figure 1 demonstrates the case where allocator 0 has pending deallocation requests for allocators 1, 2, 5 and 9, and allocator 9 has a pending request for allocator 7. For simplicity, the examples use $k = 2$, requiring only 4 buckets, and small integers as allocator addresses. In allocator 0's buckets, the requests for allocators 1, 5 and 9 have been grouped together, as their addresses share the same low 2 bits, while the request for allocator 2 has been placed in a separate bucket.

When the total size of outgoing requests reaches the configured threshold, the allocator sends the contents of each bucket to the target of the request at the head of the list, except for the allocator's *home bucket*, which is the buckets matching the relevant bits of this allocator's identifier. The contents of the home bucket are redistributed across the other buckets, but instead of using the lowest k bits, which are all identical, the next k bits of the target allocator's identifier are used. The process is repeated, alternately emptying all buckets except for the home one, and redistributing the contents of the home bucket using the next k bits. This process stops as soon as all buckets are empty, that is, after at most $\lceil N/k \rceil$ rounds, where N is the number of bits in an identifier.

Thus, when an allocator processes requests from its incoming messages queue, it may encounter requests targeted at itself as well as requests targeted at other allocators whose address shares a prefix with the current allocator's. The requests targeted at this allocator are satisfied immediately by freeing the relevant object, whereas the other requests must be forwarded. This is done by adding these requests to the allocator's outgoing buckets, just as a remote deallocation would have, to be sent to a different allocator at a later time.

For example, when allocator 0 from Figure 1 on the previous page decides to flush pending outgoing requests, it pushes the request targeted at allocator 2 onto that allocator's queue, and requests targeted at allocators 1, 5 and 9 onto allocator 9's queue. Allocator 9 will free the object from the request targeting itself and add the remaining requests to its own outgoing buckets. The state of these buckets is shown in Figure 2a on the preceding page.

When allocator 9 in turn decides to flush those requests, it first only sends the request targeted at allocator 7; it then redistributes the requests from its home bucket (indicated by the grey background), using the next 2 bits of the target allocators' addresses. The result of the redistribution is represented in Figure 2b on the previous page. Note that allocator 9's home bucket has changed because different bits of its address are now used. At this stage, and assuming 4-bit addresses, all requests in a given bucket target the same allocator and the home bucket must be empty.

The number of hops required for a message to reach its destination is bounded. On its first hop, a request will be sent to an allocator which shares the first k bits with that request's target. On the next hop, the request will be sent to an allocator which shares the first $2k$ bits, and so on. This guarantees every request eventually makes its way to the right allocator in $\lceil N/k \rceil$ hops or fewer, where N is the number of meaningful bits used by allocator addresses. On most existing 64-bit architectures, only 48 bits of address space are used. Additionally, by making allocators 2KiB aligned, we can ignore the lowest 11 address bits, leaving only 37 meaningful bits in an allocator address. Hence, given $k = 6$, at most $\lceil 37/6 \rceil = 7$ hops are necessary.

In practice however, intermediate hops are rarely necessary. `Snmalloc` is optimized to be used with one allocator per scheduler thread and at most a handful of scheduler threads per available hardware thread. Given this limited number of allocators, the probability of a collision decreases very quickly as the hops increase. Additionally, allocators are allocated contiguously from a handful of memory chunks. This means that the lower bits, which are used first, are more likely to differ across allocators than the higher bits. If fewer than 64 scheduler threads are used, i.e. fewer than the number of buckets, messages are usually sent to their destination directly. For applications that employ more threads, `snmalloc` can be reconfigured to use more buckets.

Remote Deallocation Queue The queue for remote deallocations is a lock-free queue based on the Pony language runtime [3, 4] message queue. It allows multiple producers and a single consumer.

`front` points at the first element in the queue, and `back` at the last element. The queue is always non-empty, and has a singly linked list of remote deallocations from `front` to `back`, oldest first. The code for `dequeue` only operates on `front`, and is only accessed by a single thread for a particular queue. It is given below:

```
RemoteObject* dequeue() {
  if (front.next == nullptr)
    return nullptr;
  auto first = front;
  front = front.next;
  // Acquire fence
  return first;
}
```

If `front.next` contains `nullptr`, we cannot remove an element at the moment. Otherwise, we advance the `front` pointer and return the previous front element.

The code for enqueing only modifies `back` and is accessed by multiple threads, and is given below:

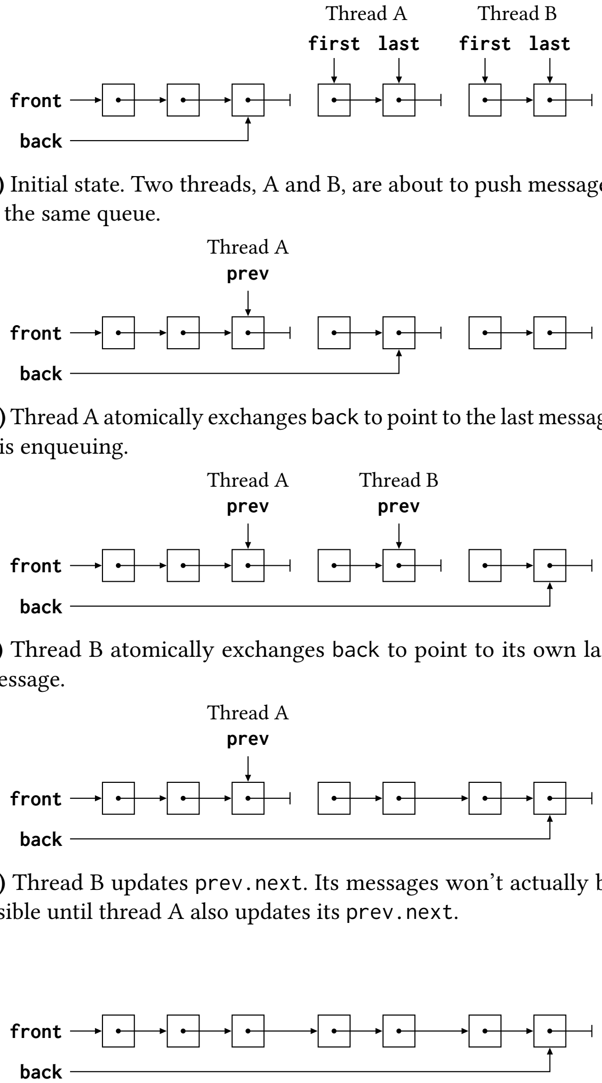
```
1 void enqueue_list(RemoteObject* first,
2                   RemoteObject* last) {
3   last.next = nullptr;
4   // Release fence
5   prev = back.exchange(last);
6   prev.next = first;
7 }
```

The enqueue uses an atomic exchange (line 5) to atomically swing the `back` pointer from the current value to the last element of the list that is being enqueued. Line 6 then links this chain of free objects into the queue.

Due to operations to grab the position in the order (5) and linking into the queue (6) happening separately, it is possible for `dequeue`s to not see `enqueue`s that have already returned, that is, the queue is not linearizable [12]. Once all `enqueue`s that were executing in parallel with a particular `enqueue` have completed then the `dequeue` will observe them. This increases performance in the queue, but may, under contended scenarios, mean reclamation is delayed.

Figure 3 on the facing page illustrates two threads pushing batches of messages to the same queue. After step c, both threads have atomically acquired the position where they will insert their messages. During step d, thread B executes line 6, linking its messages into the queue. However, because thread A acquired an earlier position in the list, thread B's messages are not visible to the receiver until thread A links its own messages, despite `enqueue_list` having completed. This finally happens in step e, making all messages visible.

To deal with weak-memory effects, we require that line 3 is visible before both lines 5 and 6. We have placed comments for the fences required. All other operations on shared state can be *relaxed* atomics.



(a) Initial state. Two threads, A and B, are about to push messages to the same queue.

(b) Thread A atomically exchanges back to point to the last message it is enqueuing.

(c) Thread B atomically exchanges back to point to its own last message.

(d) Thread B updates prev.next. Its messages won't actually be visible until thread A also updates its prev.next.

(e) Thread A updates prev.next. All messages become visible to the receiver.

Figure 3. Two threads concurrently pushing to the same deallocation queue.

2.3 Chunks, Slabs, and Allocators

Snmalloc divides the virtual address space into *chunks*. The size of chunks is configurable, but in this document we assume that it is (2^{24} bytes), and that chunks are aligned to 16MiB boundaries. In our evaluation, in Section 3, we will also consider 1MiB chunks. A chunk is either part of a *large object*, a *medium slab* or a *superslab*.

As we already said in Section 2.1, we distinguish between large, medium and small objects. Large objects require at least 16MiB, that is at least as much space as a chunk. Medium objects require between 16MiB and 64KiB, and small objects require less than 64KiB. These numbers are configurable.[1]

As we will see in the next section, given an internal pointer to an object, the page map can be used to determine whether the object is small, medium or large. In the case of a large object, the page map can also determine the size and the starting address of the object. Otherwise, further meta-data stored in the same chunk as the object is used to determine the size and start of the object, as well as the owning allocator.

Large objects are stored in one or more chunks, with their sizes rounded to the next power of two. They are handled globally using a lock-free stack per power of two.

Medium and small objects are stored in *medium slabs* and in *superslabs* respectively. Both medium slabs and superslabs are stored in precisely one chunk. The first cache-line of a medium slab or a superslab contains a pointer to the owning allocator and a description of its kind (medium or super). A medium slab contains objects of the same size only. A superslab consists of one or more *small slabs* of size 64KiB.[2] Each such small slab contains objects of the same size.

Each allocator has fast access to medium and small slabs with free space for a given size of object. Namely, all non-full medium slabs for objects of a given size owned by the same allocator are organized in a doubly-linked list. Similarly, all non-full small slabs for objects of a given size owned by the same allocator are organized in a similar list. The nodes for these lists are stored in a free object in each slab.

2.4 Page Map

Snmalloc uses a global *pagemap*, shared across all allocators, to determine the kind of each chunk. As we said earlier, the size of a chunk is configurable, but in this document we assume it is 16MiB (2^{24} bytes). The pagemap contains a single byte per chunk, each of which can be one of the following;

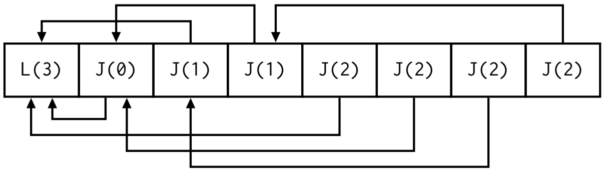
Unknown : Not managed by snmalloc.
Super : This is a superslab.
Medium : This is a medium slab.
Large(n) where $n \in [0; 24)$ [3]: This is the start of a large object consisting of 2^n chunks.
Jump(m) where $m \in [0; 24)$: This is part of a large allocation that starts at least 2^m chunks earlier in the address space.

This representation allows the pagemap to describe the entirety of 48-bit address space as a 16MiB flat map, or alternatively as a two level map with 64KiB leafs. On systems that support lazy commit (see Section 2.9 on page 8), there is no advantage in using a two-level structure because the system will provide copy-on-write zero pages for the pagemap and a zero value indicates the not-managed-by-snmalloc state.

[1] For the 1MiB chunks used in our evaluation, the boundary between small and medium objects is 16KiB, and medium and large is 1MiB.
[2] Again this is configurable.
[3] Assuming 2^{24} byte chunks and a 48-bit useable address space

It also allows the start of large objects to be found in logarithmic time in the size of that object.

For example, the pagemap entries of the 8 chunks composing a 128MiB object consist of 8 bytes; $L(3)$ (since 2^3=8) and 7 back-jumps as follows:



The back-jump entries are used to find the start of an object from its interior pointer. This is not required by the standard C interface for allocators, but can be useful for debugging.

2.5 Large Objects

Large objects require at least 16MiB, and are centrally managed. They are always aligned to a 16MiB boundary. We round the size of any large object to the next power of two, i.e. 16MiB, 32MiB, 64MiB, etc. When we allocate an object, it is always in a power of two sized address space, but we only commit the pages for the used space.[4]

We keep track of available objects for each large object size (i.e. each power of 2 greater than or equal to 16MiB) in a lock-free stack. We decommit all the pages of these blocks except for the first page, which is used as the stack entry. The lock-free stack is just a Treiber stack [21]. To handle the standard ABA issues, we use an incarnation number next to the head of the stack, and use a 128-bit compare-and-swap operation.

2.6 Size Class Calculation

Snmalloc uses a similar calculation for size classes to SuperMalloc [16]. SuperMalloc uses a 2-bit mantissa, m, and a 6-bit exponent, e, for its size classes, where this represents the size $(4 + m) * 2^{(e+1)}$. This encoding always has a leading 1. We represent our size classes differently, by making $e = 0$ mean that we don't have a leading 1, and $e > 0$ mean that we do have a leading one. We also add 4 to the exponent, ensuring that every sizeclass is a multiple of 16, and is therefore 16-byte aligned.

```
size_t sizeclass_to_size(uint8_t sc) {
  sc ++;
  auto m = sc & 3;
  auto e = sc >> 2;
  auto b = (e == 0) ? 0 : 1;
  return (m + (b * 4)) << (4 + e - b);
}
```

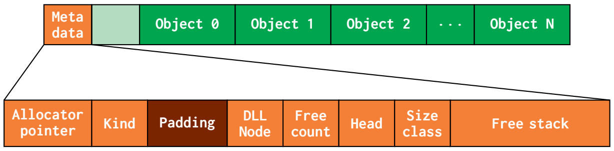
The reverse mapping cannot be pre-calculated. We use the following function:

```
uint8_t size_to_sizeclass(size_t size) {
  size --;
  auto e = 58 - clz(size | 32);
  auto b = (e == 0) ? 0 : 1;
  auto m = (size >> (4 + e - b)) & 3;
  return (e << 2) + m;
}
```

Following most bit-twiddling code for calculating binary logarithms, we subtract one, and find the highest set bit (clz, count leading zeros). By setting the 5th bit (`size | 32`), we guarantee that e is never negative. The code has a conditional statement to define b, that is compiled to a `setne` instruction on x86 processors, avoiding any branching. The correct two bits for the mantissa are selected using b to adjust the shift, accounting for a leading one or not. Finally, we can pack the exponent and mantissa into the byte.

2.7 Medium Slabs

Medium objects (size-classes from 64KiB upto 16MiB) are allocated on medium slabs. These consist of meta-data followed by medium objects. The medium objects are aligned such that the last object will abut (i.e. precisely finish at) the end of the medium slab.



The first entry in the meta-data is a pointer to the owning allocator, i.e. the allocator that owns the objects in this slab; this is required so that remote frees know which allocator to return objects to. The kind is used to distinguish medium slabs and super slabs without requiring a pagemap lookup. Because the allocator and kind are read by many threads (but written only once while the slab is allocated), we pad these in their own cache line.

The next entry in the meta-data is a doubly linked list node (128 bits) for medium slabs that have free space, and that have the same size-class and the same owning allocator. The `Free Count` (16 bits) represents how many objects on the slab are not being used. `Head` (8 bits) represents the index into the "free stack" (512 bytes) which is effectively a linked list of unused objects on this slab. The `Size-class` (8 bits) represents the rounded size of every block in this medium slab.

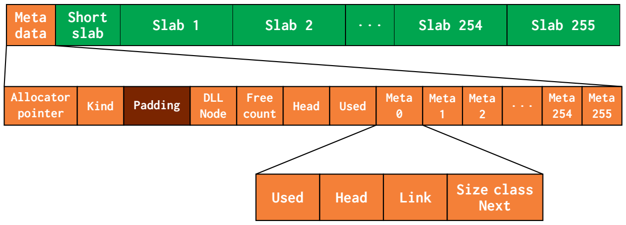
2.8 Superslabs

Small objects are stored in small slabs, which reside within superslabs. Superslabs are broken down into small slabs of 64KiB each, and there are 256 of them in a superslab. The first 64KiB of a superslab is special as it contains the meta-data

[4] We use the Windows terminology, where a *committed page* is a page of virtual memory to which some physical memory is associated whereas an *uncommitted page* is a page of virtual memory to which no physical memory is associated. See Section 2.9 on page 8 for a discussion of how this behavior varies between operating systems.

for the entire superslab, and then a *short slab* of allocations. The short slab is like any other small slab except that the starting index for allocation is after the meta-data, and it has 2KiB less space for objects than a small slab.

As with medium slabs, the last object allocated in a small or short slab will abut the end of the slab. Note that this enables short slabs and small slabs to share most of their code paths. For most allocation and deallocation operations, a short slab is simply a small slab with existing allocations that cannot be freed.



The meta-data starts with a pointer to the allocator owning the objects of the superslab. As for medium slabs, the owner and the Kind fields are padded into a separate cache-line.

The DLL Node creates a doubly-linked list of superslabs which have the same owning allocator, and which have at least one free small slab. Head is a pointer to a free small slab; it is stored as an index into the per small slab meta data indicated through Meta 0, ... Meta 255 in the diagram. Used indicates how many slabs are in use; it is encoded as twice the number of small slabs used, plus one if the short slab is used. For instance, a full superslab would have a used field of $511 = (255 * 2 + 1)$.

The meaning of the per small slab meta-data depends on whether that slab is used or not. For free small slabs, the field Used contains 0, and Size class/next contains a pointer to the next free small slab, and 0 if no such free small slab exists on the current superslab. Thus, effectively, there is a free list starting at Head using the next indexes. We now consider used slabs:

Bump Pointer-Free Lists We allocate space in small slabs through a combination of per slab free lists and bump pointer allocation. Thus all the allocated objects on a small slab will be in the space up to the *bump space*. Everything in the bump space is unallocated, but the area preceding the bump space may also contain some free elements (through calls to free).

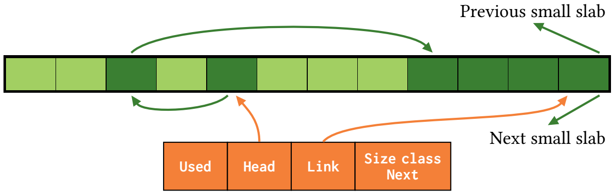
If the small slab is in use, then the field Size-class in its meta-data represents the size of the objects stored in that small slab. The field Used contains the number of allocated objects, and fields Head and Link are used to keep track of free space within the slab or across slabs. We discuss these two fields in some more detail below.

If Head is -1, then the slab is full. Otherwise, Head is either a slab-relative pointer to a free element in the small slab, or, if its bottom bit is set, it is a *bump pointer*, pointing to the start of the bump space. The free elements are represented through a linked list kept in the slab itself. This list is either terminated by a bump pointer, in which case the bottom bit is set, or, by -1 when no more bump space is left.

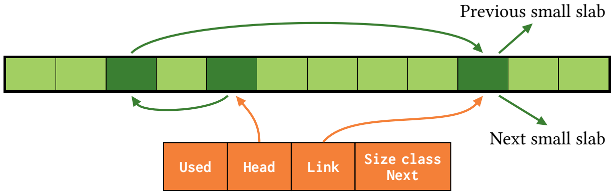
Link is used for the representation of a doubly-linked list of small slabs of the same size class which are all owned by the same allocator, and which have free space. As small slabs appear in this doubly linked list only if they have free space, we store the node for the doubly linked list in the free space in the small slab and only store a pointer to that node in the meta-data. This makes the meta-data more compact, but slightly complicates the invariants. It also means that each element needs the space for two pointers, and thus requires at least 16 bytes. When the bump space is not empty, the doubly-linked-node is the last element in the bump space. Otherwise it is the last element in the free list.

The diagrams below show a small slab (in green), and the associated meta-data (in brown). Free elements are dark green, and allocated objects are light green. In both diagrams the field Head points to the first free element in the small slab (dark green).

In the first diagram, bump space is still available, and there are two free elements in the area before it. Head points to the first free element; this contains a pointer to the second free element, which in turn contains the bump pointer. The field Link points to the last element in the bump space, which contains the doubly-linked-node.



In the second diagram the bump space has been exhausted, but there are three free elements. The field Link points to the last element in free list; this object contains the doubly-linked-node.



Because objects can be as small as the size of two pointers, the doubly-linked list node may occupy the entire object, overlapping with the free-list terminator. The block is interpreted in both ways at once. It is safe to do that because we only look at the free-list's next value during allocation, and we only allocate from the first slab in the doubly-linked list. The previous pointer in the doubly linked list has the same terminator, -1, as the exhausted bump list. This means we do not need additional checks comparing Head to Link during free-list allocation.

2.9 Portability

Snmalloc is portable across the Windows, Linux, macOS, and FreeBSD operating systems, running in both the FreeBSD kernel and userspace. We maintain this portability both to avoid accidentally depending on features that are specific to a single operating system and to provide insight into the OS features that make it difficult or easy to write a modern high-performance allocator.

The largest difference between our supported operating systems is support for *lazy commit*. Most POSIX operating systems (including macOS, FreeBSD, and Linux) reserve address space and then lazily provide physical pages the first time they are written to by userspace software. In contrast, on Windows, which does not support lazy commit, virtual address space is *reserved* and then explicitly *committed* to provide physical pages as backing storage. This has the advantage that memory exhaustion happens at the allocation site (where it may be recoverable), not later when memory is used, but the disadvantage that physical memory is consumed more aggressively. FreeBSD provides a similar abstraction to code running in kernel space, where address space is reserved using vmem[5] and then committed with kmem_back. Supporting the Windows model made it easy to port the allocator to run inside a kernel, where the environment has very different constraints.

Looking up an address in the page map, as described in Section 2.4 on page 5, requires finding the start of the chunk, which in turn requires chunks to be strongly aligned. Both Windows and FreeBSD provide userspace APIs to allocate such memory regions, via VirtualAlloc2, and mmap with the MAP_ALIGNED attribute, respectively. On Linux and macOS, snmalloc initially requests a larger region and then returns the parts that are outside of the aligned range.

FreeBSD and Linux both provide an madvise MADV_FREE hint, that allows the kernel to lazily discard the underlying mapping. This is closely related to lazy commit. Linux's madvise MADV_DONTNEED resets a mapping to the lazy commit state; the same semantics are achieved on any POSIX platforms by calling mmap over the same region. In contrast, the madvise MADV_FREE hint allows the kernel to revert a mapping to the lazy commit state any time between the madvise call and the next write to the page, but does not require it to do so. On Windows, we implement something similar in userspace the low memory notifications. Snmalloc does not return physical memory to the kernel until notified that memory is scarce. On the first allocation or deallocation after this event, snmalloc attempts to decommit unused slabs, returning the physical pages for reuse. This avoids any system calls or page-table updates while memory is plentiful at the cost of some very high-overhead operations when memory becomes scarce.

On systems without lazy commit, snmalloc has a slightly higher overhead because it returns physical pages to the OS once a chunk is no longer used and then reacquires them when reallocating memory. In contrast, on systems with lazy commit, the allocator returns the memory to the platform, which lazily allocates physical pages when the memory is used. This approach means that programs that make large allocations but touch only a small part of them will consume more physical memory on operating systems without lazy commit.

3 Evaluation

The performance of any allocator is critical information used to justify design decisions. Initially, we evaluate the performance and memory overhead of snmalloc through micro-benchmarks, meant to simulate different allocation patterns. Next, we explore the effect of several design decisions and how performance is impacted. Finally, snmalloc is integrated into real-world applications and we measure the performance before and after the integration.

We compare the performance of snmalloc with popular existing allocators: Hoard (git-e52d3e0)[6], jemalloc (5.1.0), lockfree (git-915f51b)[7], lockless[8], ptmalloc2 (from glibc 2.27), rpmalloc (git-eee5329)[9], scalloc (1.0.0)[10] SuperMalloc (git-709663f)[7], tbbmalloc (from TBB 2019 Update 3) and tcmalloc (from gperftools 2.7).

We have also compared snmalloc configured to use both 16MiB and 1MiB chunks. The 1MiB chunk configuration, snmalloc-1mib, was developed for virtual address space constrained scenarios, such as 32-bit architectures. When benchmarking we found neither 1MiB nor 16MiB chunks to be superior for all workloads. We include both to show there is still opportunity to further tune snmalloc.

3.1 Micro-benchmarks

We start our evaluation by comparing the performance and memory usage of snmalloc against other allocators across two micro-benchmarks. While not a perfect representation of real-world usage, these benchmarks provide a good way to explore the strengths and weaknesses of each allocator. For each benchmark, we report the average of 5 runs, and represent the minimum and maximum values as error bars. The benchmarks were run on an Ubuntu 18.04 *Standard F64s_v2* (64 vcpus[11], 128 GB memory) Azure virtual machine.

Symmetric Workload We designed a benchmark to simulate a symmetric workload, where each thread repeatedly

[5] vmem is a general allocator for contiguous ranges of numbers and can be used to allocate regions within virtual address spaces.

[6] We experienced segmentation faults when using the latest Hoard release. These did not occur when using the latest git revision.

[7] No stable release available

[8] No version or revision number available

[9] Includes minor build system fixes over the last release, 1.3.1

[10] We experienced occasional segmentation faults when using scalloc, but were able to produce enough results to include it.

[11] Dual-socket *Intel Xeon Platinum 8168*

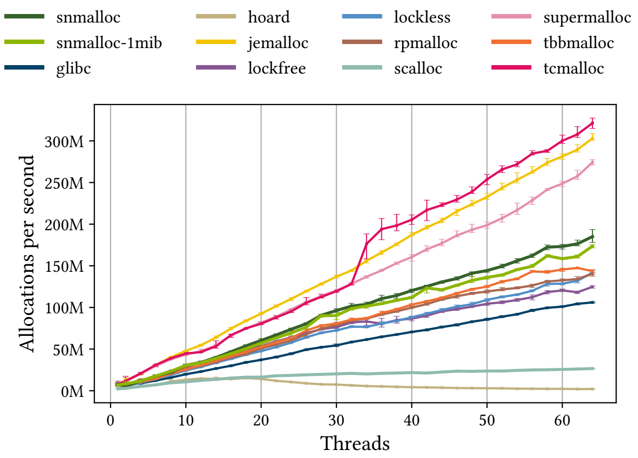


Figure 4. Allocation throughput of different allocators on a symmetric workload.

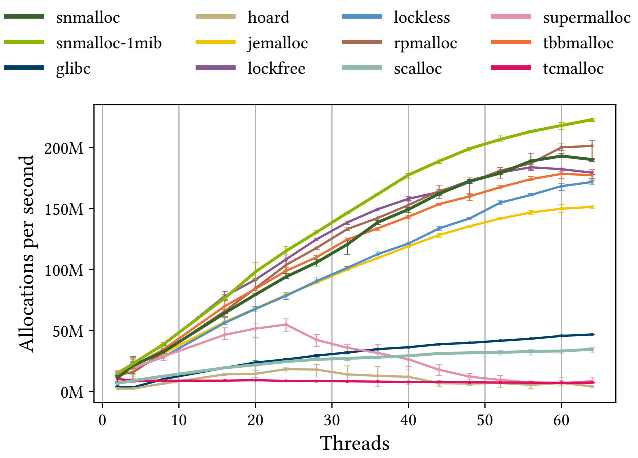


Figure 5. Allocation throughput of different allocators on a producer/consumer workload.

allocates an object, swaps it with another thread, and deallocates the received object. Figure 4 shows the throughput of different allocators over a varying number of threads.

This is the scenario for which thread-caching allocators are optimized. Individual threads place the objects they have freed in their local caches and reuse them when allocating an object of the same size class. There is no need for any form of synchronization with the thread that originally allocated this object. This explains the good performance of jemalloc, tcmalloc and SuperMalloc on this benchmark.

Snmalloc, on the other hand, always returns deallocated objects to their originating allocator, even if the object could have been reused later. While reasonably lightweight, message passing still represents a non-negligible synchronization cost. Nevertheless, snmalloc's performance remains better than most allocators.

Producer/Consumer The second allocation pattern that we consider is a producer/consumer scenario, where producer threads only allocate objects and consumer threads only deallocate them.

The malloc-test benchmark, which we obtained from the SuperMalloc [16] repository, reproduces an asymmetric producer/consumer scenario. Half the threads behave as producers, while the other half behave as consumers. Each producer thread repeatedly allocates 4096 objects and stores the pointers to the objects in a message. The messages are pushed onto a global lock-protected queue. Consumer threads remove a single message from the global queue and deallocate all the objects it contains. The benchmark can be either configured to use a single object size, or to randomly pick a different size on every allocation, ranging from 8 to 2048 bytes.

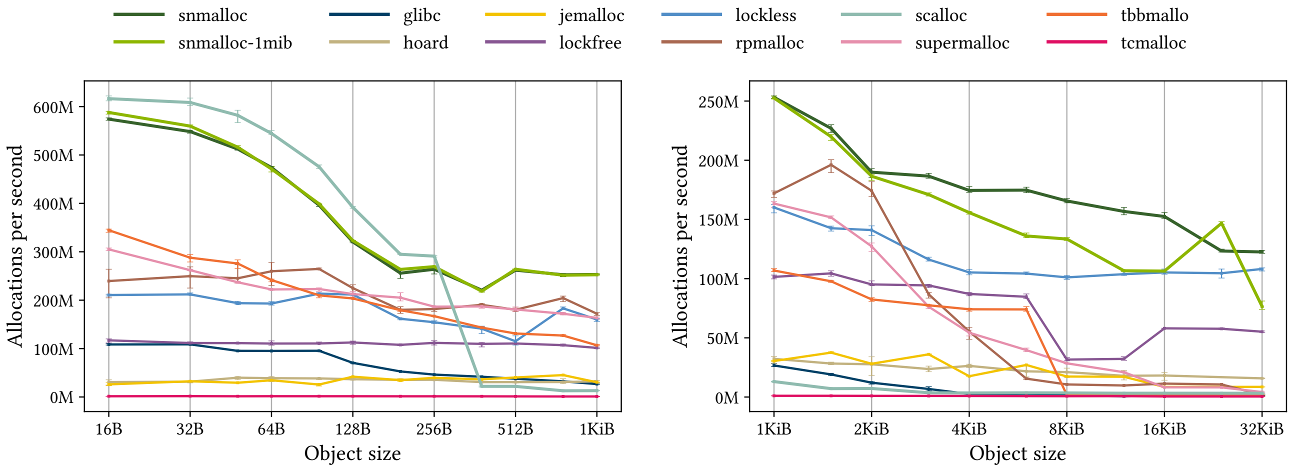
The allocation throughput of each malloc implementation, when running the benchmark with randomized object sizes, can be seen in Figure 5, using the total number of worker threads as the x-axis.

Unlike the symmetric workload scenario, thread-caching allocators perform quite badly in this benchmark. The consumer threads' caches rapidly become full and their contents must be returned to their original heaps. Conversely, the caches of the producer threads are depleted, and must be filled from global heaps. Synchronization must occur between threads whenever objects are returned to, or fetched, from the shared heaps. Given the relative complexity of the heap data structures, this synchronization is usually expensive, using either locks or compare-and-swap loops. Tcmalloc performs particularly poorly in this scenario, whereas it excelled in the symmetric workload case.

This is one of the use cases for which snmalloc was designed to excel. Via message passing, consumer threads can cheaply return deallocated objects to their original thread. The synchronization required is limited to a single atomic exchange per batch of deallocated objects.

Figure 6 on the following page shows the results of running this benchmark for different sized allocations. The x axis is the size of the objects in bytes, the y axis is the number of allocations per second. Snmalloc achieves surprisingly good performance on extremely small objects, but throughput quickly drops as the size increases, until around 256 bytes, where it becomes reasonably stable.

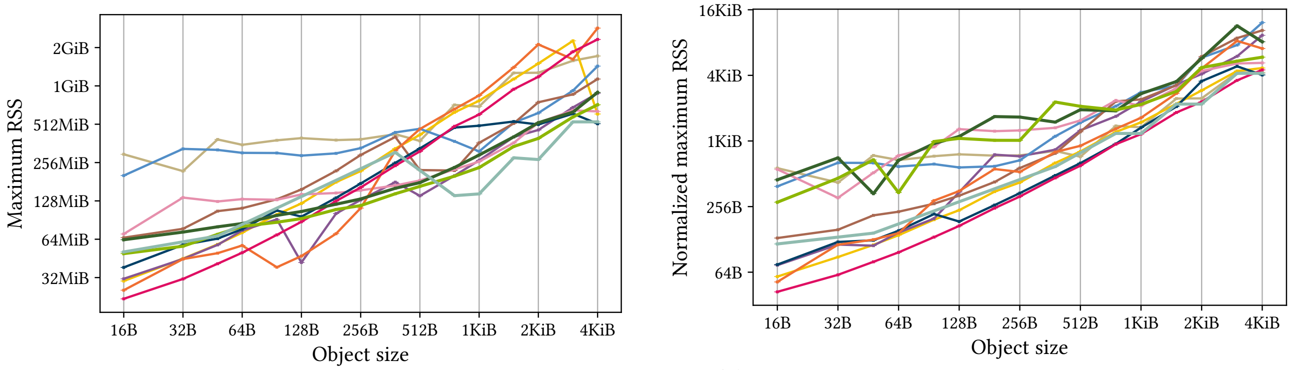
Space Overheads Figure 7a on the next page shows the maximum memory allocated during the producer/consumer benchmark, for each allocator. We modified the benchmark so that the producers touched all of the memory, guaranteeing that physical pages were allocated by the OS. A perfect allocator would use precisely as much memory as the application requested. This is somewhat complicated for a multi-threaded workload because the throughput of the allocator can significantly impact the total number of live objects. In this producer/consumer benchmark, allocators that can free memory faster than they can allocate memory will have low



(a) Object sizes ranging from 16B to 1KiB.

(b) Object sizes ranging from 1KiB to 32KiB.

Figure 6. Allocation throughput of different allocators on a producer/consumer workload, for various object sizes. For readability, the two plots use different y-axis scales.



(a) Absolute maximum resident set size.

(b) Maximum resident set size, normalized by maximum queue length.

Figure 7. Maximum resident set size when running the producer/consumer benchmark with 64 worker threads

peak memory usage, whereas implementations that allocate faster than they deallocate will see high overhead.

Total overhead depends on two additional factors. First, the meta-data overhead for the allocator, which is independent of concurrency and depends on the amount of extra information that the allocator stores per object. Second, an allocator that performs thread-caching will artificially prolong object lifetimes by maintaining a small list of allocations that are "free" but not globally available for reuse. `Snmalloc`'s message queues keep objects live until they are returned to their owning thread.

To discover how the allocation rate affects total memory usage, we instrumented the benchmark to measure the peak size of the queue that connects producer to consumers threads. Figure 7b shows the result of normalizing peak memory usage against peak queue length. `Snmalloc` has similar overheads to Lockless and SuperMalloc.

Warmup Time While evaluating performance, we noticed SuperMalloc performed significantly better when running benchmarks for longer periods of time. It would have been too time consuming to run every benchmark long enough for all allocators to warm up; we have instead used a 20 second run-time for all results of the producer/consumer benchmark presented here.

Nevertheless, to ensure our results are not unfair because of too short run-times, we ran the producer/consumer benchmark over increasing durations, the results of which are presented in Figure 8 on the next page. These results use the same configuration as in Figure 5 on the preceding page, using 64 workers, which is the machine's capacity.

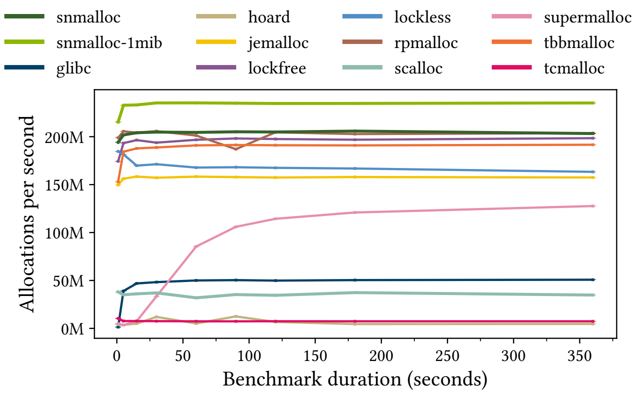


Figure 8. Allocation throughput of different allocators on a producer/consumer workload, for increasing benchmark durations

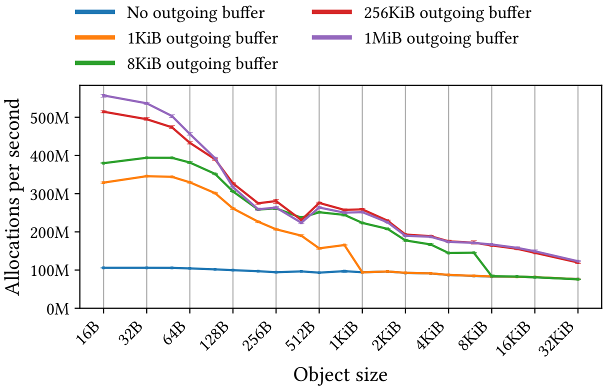


Figure 9. Allocation throughput of `snmalloc`, using different outgoing buffer capacities.

While glibc's allocator and rpmalloc require a bit of time to warm up, they are at their peak performance by 20 seconds. SuperMalloc on the other hand requires much longer, over 100 seconds, to reach peak throughput. Even then however, it performs worse than `snmalloc`.

We believe the long warm-up time is caused by SuperMalloc's hierarchical structure. It employs multiple levels of caching, per-thread, per-CPU and a global cache, with increasing amounts of synchronization to access each level. Objects deallocated by consumer threads travel through each object cache level before they reach producer threads. Just after program startup, all caches are empty and producers cannot reuse the deallocated objects. After warm-up, every level of the object cache is full, improving performance.

3.2 Batched Message Passing

Batching of remote deallocations, as described in Section 2.2 on page 2, is a key element of `snmalloc`'s design. It significantly reduces the number of atomic exchanges necessary to send messages, at the cost of an increase in memory usage, caused by the increase in the duration between the call to `free` and the actual deallocation of the object.

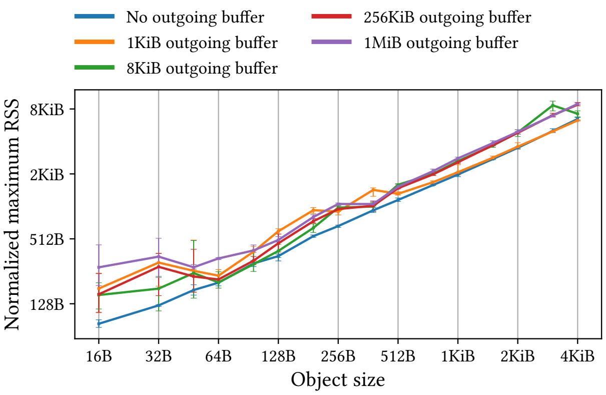


Figure 10. Normalized maximum resident set size when using `snmalloc` with different outgoing buffer capacities.

`Snmalloc` uses a configurable threshold to determine when pending outgoing deallocation requests should be sent. Figure 9 shows the throughput of `snmalloc` built with different threshold values. It also shows the throughput with batching entirely disabled; this causes every call to `free` with a remote object to perform an atomic exchange.

Unsurprisingly, disabling batching has a large negative impact on performance. Even a small outgoing buffer size, 1KiB per allocator, is sufficient to improve performance. It becomes ineffective again when a single object is larger than the threshold. We have found 1MiB to give the best results. Larger sizes provide very little to no improvement; message passing is not the main bottleneck beyond that point.

Because the reclamation of deallocated objects is delayed, the improved performance brought by batching comes at the cost of higher memory consumption. Figure 10 shows the normalized memory consumption, as described previously, when using `snmalloc` with various outgoing buffer capacities.

3.3 Real Applications

Micro-benchmarks are a useful tool to discover and demonstrate the strengths and weaknesses of `snmalloc`, but are not the most important measure. The most critical test of `snmalloc` is its performance in real-world systems and the effort required to modify them to use `snmalloc`. We are able to provide only circumstantial evidence of developers using `snmalloc`; however, we modified two real world applications and compared their performance before and after.

FaRM is a main memory distributed computing platform [6, 7, 20], deployed by Microsoft as part of the Bing search engine [19]. We modified the FaRM source code to be able to run with 3 different allocation strategies: using `snmalloc`, using Rockall, the internal sizeclass based allocator built

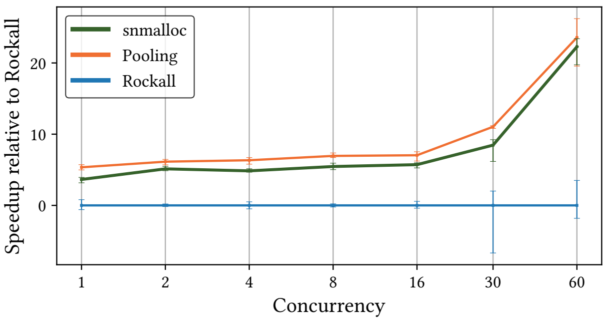


Figure 11. Relative performance increase in FaRM using snmalloc and the pooling allocator, compared to Rockall

into FaRM, and finally employing memory pooling for all memory allocations on the critical path. The pooling version avoids memory allocation on the critical path by using memory pools sized based on the application's needs.

To benchmark the different allocation strategies we used FaRM's implementation of the Yahoo! Cloud Serving Benchmark [5]. This benchmark was designed to represent common usage patterns of key-value stores employed by online services. We used a 50% mix of key-value reads and updates to the key-value store. The store had 16 byte keys and 1024 byte values, and is implemented as a B-tree stored in 3 way replicated FaRM memory on a cluster of 5 machines.

Figure 11 shows a comparison of the performance using snmalloc versus using Rockall or memory pooling. We observe a performance increase of 3.5–22% when comparing snmalloc to the base allocator in FaRM and a slowdown of 1–2.5% when compared to pooling. The pooling implementation is carefully tuned to the workload, whereas snmalloc is a general-purpose allocator. For an application such as FaRM, deployed at scale in a cloud environment, creating hand optimized memory pools to gain a 5–23% performance improvement over existing general purpose allocators can be worthwhile, even though the implementation effort is significant, that is not the case for many programs. Using snmalloc versus custom pools has a small performance overhead of 1–2.5% in FaRM, with no implementation effort required. We therefore believe that snmalloc is an attractive alternative to writing hand optimized memory pools and allocators.

SPECspeed Integer The SPECspeed Integer suite is a collection of benchmarks from the SPEC CPU 2017 benchmarking tool. It compares system performance by measuring the time taken to run real-world workloads. These benchmarks are mostly single-threaded and so do not take advantage of snmalloc's message passing design.

Figure 12 shows the results of running the suite using the different memory allocators. The results are shown relative to snmalloc's performance, as the range of results varies wildly between benchmarks. The suite was run on an Ubuntu 18.04 *Standard D3_v2* (4 vcpus[12], 14 GB memory) Azure virtual machine. As for the micro-benchmarks, we report the average of 5 runs, and represent the minimum and maximum values as error bars.

The results show that snmalloc is well suited as a general purpose allocator, performing as well or better than the widely used jemalloc, tcmalloc and glibc's allocator on all benchmarks. Other results are within error margin, with the exception of SuperMalloc and lockfree which perform better than snmalloc on some of the benchmarks.

We have also run the SPECspeed suite with its *test* data set, which runs the benchmarks with different, smaller inputs. The results of this suite are shown in Figure 13. While this data set should not be used to report official SPEC results, it provides an interesting perspective on the performance of allocators on shorter workloads. The results coincide with our earlier results on warm-up time in Section 3.1 on page 8, and suggest that snmalloc will have significantly better performance than many other allocators for short-lived command-line applications and remain competitive for long-running workloads.

The SPECspeed Integer suite includes 5 more benchmarks not represented here. They focus heavily on computational tasks, such as video (625.x264_s) or data (657.xz_s) compression. These carefully avoid performing a large number of allocations in performance-critical sections and, as a result, we have observed only a small difference between the various allocators.

Usability of a new memory allocator is notoriously difficult to measure. To obtain an estimate, we investigated the difficulty of replacing different codebases' memory allocators with snmalloc: First, we replaced the default memory allocator with snmalloc in FaRM. No changes were required in snmalloc but 23 new lines of C++ were needed in FaRM. This replacement involved only a few hours of work by a person unfamiliar with the FaRM codebase.

Next, we modified the FreeBSD [17] C standard library (libc) from the 12.0 release to use snmalloc, replacing jemalloc as the operating system's default malloc implementation. This replacement did not involve any change to C/C++ code, and required the addition of only 24 lines to a Makefile. Finally, a build-time option to switch between jemalloc and snmalloc required the addition of only 6 lines to a makefile.

The above observations indicate that replacing allocators by snmalloc in existing projects, as well as the use of snmalloc in new projects, incurs very low overhead in terms of developer time.

The replacement of jemalloc with snmalloc also gave us some information as to relative binary sizes in similar conditions. Our baseline FreeBSD libc, using jemalloc, is 2,052,560 bytes, whereas the version using snmalloc is only 1,701,112 bytes. The snmalloc.o file that is linked into libc is 418,576

[12] *Intel Xeon CPU E5-2673*

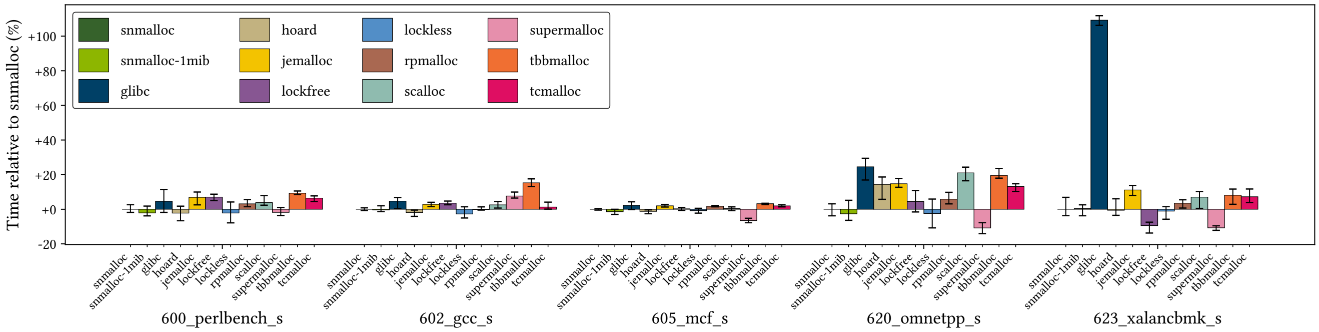


Figure 12. Performance of different allocators on the SPECspeed Integer suite, using the `ref` data set.

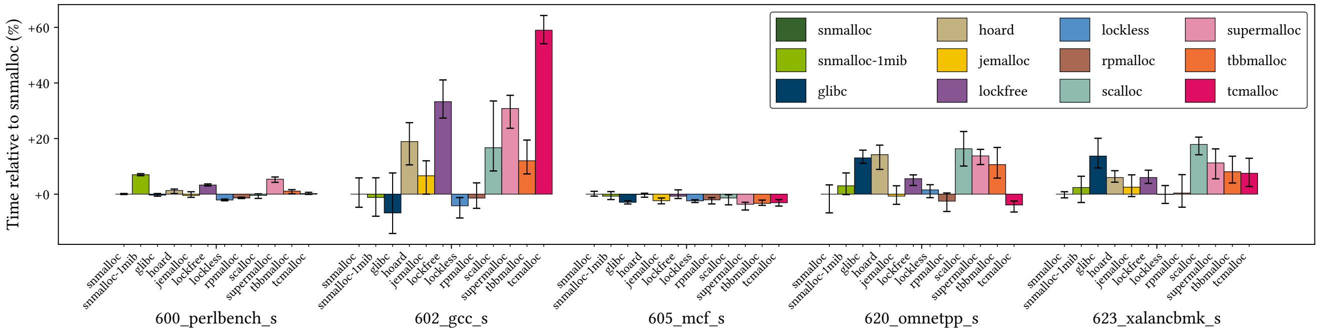


Figure 13. Performance of different allocators on the SPECspeed Integer suite, using the `test` data set.

bytes, so `snmalloc` accounts for approximately 24.6% of the total binary size. In contrast, jemalloc contributes 37.5% to the total size of a mature POSIX and C11 standard library implementation.

4 Related Work

`Snmalloc` differentiates itself from existing allocators by its efficient message dispatching scheme and compact slab metadata. However many of its other design points draw inspiration from previous work. In this section we review some of these works, and compare them with `snmalloc`.

Slabs of Uniformly Sized Objects Many existing allocators use slabs composed of a single size of object, albeit with different terminology (eg. runs, spans, superblocks or chunks). Whereas `snmalloc` uses two different slab size, for small and medium objects, Hoard, SuperMalloc, rpmalloc and tbbmalloc all use a unique slab size.

scalloc reserves the same amount of address space for each slab, in the form of *virtual spans*, but only uses part of this space, the *real span*, whose size depends on the size-class of the contained objects. This allows an empty virtual span to be reused for a different size-class, while keeping the benefit of specialized slab size.

Owned Heaps rpmalloc, tbbmalloc, lockless and scalloc have used a similar design as `snmalloc`, making each thread own an individual allocator. In order to support remote deallocations, released objects are sent to the original allocating thread, using either a Treiber stack or an MPSC queue similar to ours.

However, none of the existing allocator batch deallocation requests the way `snmalloc` does. tbbmalloc, lockless and scalloc use per-slab queues rather than per allocator to distribute the memory traffic on those queues. This increases the amount of per-slab meta-data required by not just one pointer, but by an entire cache-line in order to avoid false-sharing.

Shared Heaps An alternative to owned heaps is to have threads share access to the heaps. SuperMalloc and tcmalloc use a single global one, whereas Hoard, jemalloc and glibc's allocator use multiple ones, proportionally to the number of available core. Threads choose a heap to allocate from either by hashing their thread identifier, or through round-robin assignment.

Accessing the heaps must be synchronized to protect against concurrent access, usually through mutexes or spinlocks. Using multiple heaps helps reduce contention, but remote deallocations will still cause multiple threads to require access to one heap. Allocators with shared heaps therefore use per-thread object caches, in the shape of size-specific free lists. These caches are unsynchronized and very fast to access. Deallocated are placed in the caches for later reuse instead of being returned to their original heap.

In addition to the per-thread cache, SuperMalloc uses both per-CPU caches and a global cache. Because only one thread can access the per-CPU cache at a time, the access is very likely to be uncontended. Contention at this level of cache can only happpen if the thread is preempted in the middle of an allocation, or if the thread was migrated from one CPU to another, between the time it queried the current CPU's identifier and the time it accesses the cache.

Free Space Tracking Many allocators use a combination of free list and bump allocation to maintain the set of unallocated objects, similar to `snmalloc`'s small slabs.

SuperMalloc, rpmalloc, tbbmalloc, scalloc, Hoard use separate pointers for the head of the list and the start of the bump allocation area. lockless appears to use a similar data structure as `snmalloc`, where the terminator of the free list acts as the bump pointer. It's per-slab meta-data is not as compact as `snmalloc`'s however, as it uses absolute pointers rather than smaller slab-relative ones, and the doubly linked list node, used to link slabs together, is stored in the slab's meta-data rather than in a free object.

5 Conclusion

We have presented `snmalloc`, a new allocator in the allocator/deallocator design space. It is based on a novel message passing scheme which returns deallocated objects to the owning allocator in batches. To deal with the dynamic nature of the number of allocators, we developed a novel dispatching scheme combining ideas from hashing and temporal radix trees. As customary in allocation/deallocation work, we use slabs to store objects of the same size; we use meta-data and a novel combination of free-lists and bump allocation to keep track of available space.

Evaluations results so far are very encouraging. Comparing `snmalloc` with most current allocators tends to show that `snmalloc`'s performance is comparable in symmetric workloads and has significant advantages in producer/consumer ones. The most important criterion though, is performance when incorporated into real software. There, the results tend to show that `snmalloc`'s performance is superior. Incorporating `snmalloc` into existing applications was easy, opening the door to wider adoption. However, our experience is limited, as so far we have only experimented with FaRM and SPEC CPU2017. We plan further evaluations and improvements.

`Snmalloc` is available at `https://github.com/Microsoft/snmalloc`.

References

[1] Martin Aigner, Christoph M. Kirsch, Michael Lippautz, and Ana Sokolova. 2015. Fast, Multicore-Scalable, Low-Fragmentation Memory Allocation through Large Virtual Memory and Global Data Structures. *CoRR* abs/1503.09006 (2015). arXiv:1503.09006 http://arxiv.org/abs/1503.09006

[2] Emery D. Berger, Kathryn S. McKinley, Robert D. Blumofe, and Paul R. Wilson. 2000. Hoard: A Scalable Memory Allocator for Multithreaded Applications. *SIGOPS Oper. Syst. Rev.* 34, 5 (Nov. 2000), 117–128. https://doi.org/10.1145/384264.379232

[3] Sylvan Clebsch. 2018. Pony: co-designing a type system and a run-time. https://spiral.imperial.ac.uk/handle/10044/1/65656 PhD thesis, Imperial College London.

[4] Sylvan Clebsch, Sophia Drossopoulou, Sebastian Blessing, and Andy McNeil. 2015. Deny capabilities for safe, fast actors. In *Proceedings of the 5th International Workshop on Programming Based on Actors, Agents, and Decentralized Control.* ACM, 1–12.

[5] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing*. ACM, 143–154.

[6] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. 2014. FaRM: Fast remote memory. In *11th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 14)*. 401–414.

[7] Aleksandar Dragojević, Dushyanth Narayanan, Edmund B. Nightingale, Matthew Renzelmann, Alex Shamis, Anirudh Badam, and Miguel Castro. 2015. No Compromises: Distributed Transactions with Consistency, Availability, and Performance. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP '15)*.

[8] Jason Evans. 2006. A Scalable Concurrent malloc(3) Implementation for FreeBSD. In *BSDCan*.

[9] Jason Evans. 2019. jemalloc memory allocaror. http://jemalloc.net/

[10] Sanjay Ghemawat and Paul Menage. 2018. TCMalloc: Thread-Caching Malloc. http://goog-perftools.sourceforge.net/doc/tcmalloc.html

[11] Wolfram Gloger. 2006. Wolfram Gloger's malloc homepage. http://www.malloc.de/en/

[12] Maurice P Herlihy and Jeannette M Wing. 1987. Axioms for concurrent objects. In *Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. ACM, 13–26.

[13] Lockless Inc. 2012. Optimization Tricks used by the Lockless Memory Allocator. https://locklessinc.com/articles/allocator_tricks

[14] Jansson, Mattias. 2019. Rampant Pixels Memory Allocator. https://github.com/rampantpixels/rpmalloc

[15] Alexey Kukanov and Michael J Voss. 2007. The Foundations for Scalable Multi-core Software in Intel Threading Building Blocks. *Intel Technology Journal* 11, 4 (2007).

[16] Bradley C Kuszmaul. 2015. SuperMalloc: A Super Fast Mulithreaded Malloc for 64-bit Machines. In *ISMM*.

[17] Marshall Kirk McKusick, George Neville-Neil, and Robert N.M. Watson. 2014. *The Design and Implementation of the FreeBSD Operating System* (2nd ed.). Addison-Wesley Professional.

[18] Maged M Michael. 2004. Scalable lock-free dynamic memory allocation. In *ACM Sigplan Notices*, Vol. 39. ACM, 35–46.

[19] Alex Shamis and Dushyanth Narayanan. [n. d.]. Tech Showcase: FaRM - Microsoft Research. https://www.microsoft.com/en-us/research/video/farm/

[20] Alex Shamis, Matthew Renzelmann, Stanko Novakovic, Georgios Chatzopoulos, Aleksandar Dragojevic, Dushyanth Narayanan, and Miguel Castro. 2019. Fast General Distributed Transactions with Opacity. ACM Association for Computing Machinery.

[21] R. K. Treiber. 1986. Systems programming: Coping with parallelism. International Business Machines Incorporated, Thomas J. Watson Research Center.