Mimalloc: Free List Sharding in Action

Microsoft Technical Report MSR-TR-2019-18, June 2019.

DAAN LEIJEN, Microsoft Research
BENJAMIN ZORN, Microsoft Research
LEONARDO DE MOURA, Microsoft Research

Modern memory allocators have to balance many simultaneous demands, including performance, security, the presence of concurrency, and application-specific demands depending on the context of their use. One increasing use-case for allocators is as back-end implementations of languages, such as Swift and Python, that use reference counting to automatically deallocate objects. We present mimalloc, a memory allocator that effectively balances these demands, shows significant performance advantages over existing allocators, and is tailored to support languages that rely on the memory allocator as a backend for reference counting. Mimalloc combines several innovations to achieve this result. First, it uses three page-local sharded free lists to increase locality, avoid contention, and support a highly-tuned allocate and free fast path. These free lists also support *temporal cadence*, which allows the allocator to predictably leave the fast path for regular maintenance tasks such as supporting deferred freeing, handling frees from non-local threads, etc. While influenced by the allocation workload of the reference-counted Lean and Koka programming language, we show that mimalloc has superior performance to modern commercial memory allocators, including tcmalloc and jemalloc, with speed improvements of 7% and 14%, respectively, on redis, and consistently out performs over a wide range of sequential and concurrent benchmarks. Allocators tailored to provide an efficient runtime for reference-counting languages reduce the implementation burden on developers and encourage the creation of innovative new language designs.

Additional Key Words and Phrases: Memory Allocation, Malloc, Free List Sharding

1. INTRODUCTION

Modern memory allocators have to balance many simultaneous demands, including performance, security, parallelism, and application-specific demands depending on the context of their use. One increasing use-case for allocators is as back-end implementations of languages, such as Swift (Wilde et al., 2011), that use reference counting to automatically deallocate objects, or like Python (Sanner and others, 1999), that typically allocate many small short-lived objects.

When developing a shared runtime system for the Lean (Moura et al., 2015) and Koka (Leijen, 2017, 2014) languages, these two use cases caused issues with current allocators. First of all, both Lean and Koka are functional languages that perform many small short-lived allocations. In Lean, using a custom allocator for such small allocations outperformed even highly optimized allocators like jemalloc (Evans, 2006). Secondly, just like Swift and Python, the runtime system uses reference counting (Ullrich and Moura, 2019) to manage memory. In order to limit pauses when deallocating large data structures, we also need to support deferred decrementing of reference counts. To do this well, cooperation from the allocator is required – as the best time to resume a deferred decrement is when there is memory pressure.

To address these issues, we implemented a new allocator that uses various novel ideas to achieve excellent performance:

- The main idea is to use extreme *free list sharding*: instead of one large free list per size class, we instead have a free list per mimalloc *page* (usually 64KiB). This keeps locality of allocation as `malloc` allocates inside one page until that page is full, regardless of where other objects are freed in the heap.
- Moreover, we use separate *thread-free* lists for frees by other threads to avoid atomic operations in the fast path of `malloc`. These thread-free lists are also sharded per page to minimize

contention among them. Such list is moved to the local free list atomically every once in a while which effectively *batches* the remote frees (Liétar et al., 2019).

- Finally, we use a third *local-free* list per page for thread-local frees. When the allocation free list becomes empty, the local-free list becomes the new free list. This design ensures that the generic allocation path is always taken after a fixed number of allocations, establishing a *temporal cadence*. This routine can now be used to amortize more expensive operations: 1) do free-ing for deferred reference count decrements, 2) maintain a deterministic heartbeat, and 3) collect the concurrent thread-free lists. Using the separate local-free list thus enables us to have a single check in the fast allocation path to handle all the above scenarios through the generic "collection" routine.
- We highly optimize the common allocation and free code paths and defer to the generic routine in other cases. This means that the data structures need to be very regular in order to minimize conditionals in the fast path. This consistent design also reduces special cases and increases code reuse – leading to more regular and simpler code. The core library is less than 3500 LOC, much smaller than the core of other industrial strength allocators like tcmalloc (~20k LOC) and jemalloc (~25k LOC).
- There are no locks, and all thread interaction is done using atomic operations. It has bounded worst-case allocation times, and meta-data overhead is about 0.2% with at most 16.7% ($\frac{1}{8}$th) waste in allocation size classes.

We tested mimalloc against many other leading allocators over a wide range of benchmarks and mimalloc consistently outperforms all others (Section). Moreover, we succeeded to outperform our own custom allocators for small objects in Lean. Our results show that mimalloc has superior performance to modern commercial memory allocators, including tcmalloc and jemalloc, with speed improvements of with speed improvements of 7% and 14%, respectively, on redis, and consistently out performs over a wide range of sequential and concurrent benchmarks with similar peak memory usage.

Historically, allocator design has focused on performance issues such as reducing the time in the allocator, reducing memory usage, or scaling to many concurrent threads. Less often, allocator design is primarily motivated by improving the reference locality of the application. For example VAM (Feng and Berger, 2005) and early versions of PHKmalloc also use free list sharding to ensure that sequential allocations often come from the same page. mimalloc also improves application memory reference locality and improves on VAM by implementing multi-threading and adding additional sharded free lists to reduce contention and support amortizing maintenance tasks. Our design demonstrates that allocators focused on improving application memory locality can also provide high allocator performance and concurrent scalability.

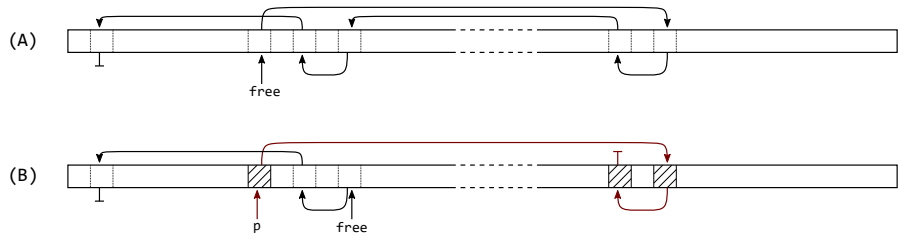
In the rest of this paper, we present the design of mimalloc, including motivating the three free lists, consider issues such as security and portability, and evaluate its performance against many state of the art allocator implementations. mimalloc is implemented in C, and runs on Linux, FreeBSD, MacOSX, and Windows, and is freely available on github (Leijen, 2019b), and with its simplified and regular code base, is particularly amenable to being integrated into other language runtimes.

2. FREE LIST SHARDING

We start with an overview of the specifics of free list sharding, the local free list, and the thread free list. After this, Section 3 goes into the details of the full heap layout (Figure 1) and the implementation of `malloc` and `free`, followed by the benchmark results in Section 4.

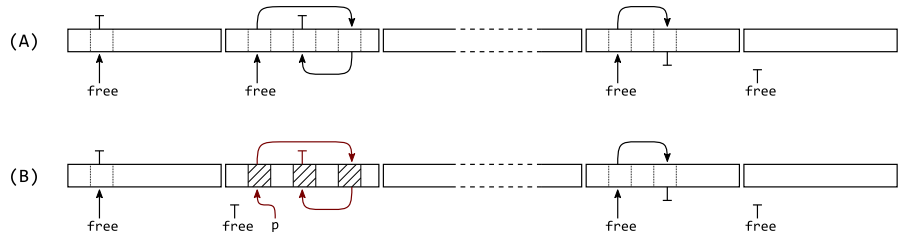
2.1. The Allocation Free List

The allocation pattern for functional style programming is to allocate and free many small objects. Many allocators use a single free list per size class which can lead to bad spatial locality where objects belonging to a single structure can be spread out over the entire heap. Consider for example the following heap state (A) where the free list spans a large part of the heap:



When allocating a list `p` of three elements, we end up in state (B) where the newly allocated list is also spread out over a large part of the heap with bad spatial locality. This is not an uncommon situation. On the contrary, most functional style programs will converge to this form of heap state. This happens in particular when folding over older data structures and building new data structures of a different size class where the interleaved allocation leads to these spread-out free lists.

To improve the spatial locality of allocation, mimalloc use *free list sharding* where the heap is divided into *pages* (per size-class) with a free list per page (where pages[1] are usually 64KiB for small objects). The previous heap state will now look like following situation (A), where each page has a small free list:



After allocating the three element `p` list, we end up in state (B) where the list is now fully allocated within the page with much better spatial locality. We believe that the good performance of mimalloc comes in a large part from the improved allocation locality.

To test this, we did an experiment in the Lean compiler (Moura et al., 2015). Version 3 of the compiler had a custom allocator for allocating small objects where it used a single free list. We replaced this implementation with just a sharded free list per slab (page) and on some benchmarks with large data structures in a 1GiB heap, we saw performance improvements of over 25% with this single change! Early work by Feng and Berger (2005) on the locality improving VAM allocator also used a sharded free list design and they measured a significant reduction in the L2 cache misses.

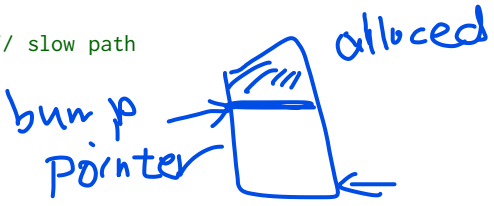
2.2. No Bump Pointer

The allocation path for allocating inside a page can now simply pop from the page local list:

```
void* malloc_in_page( page_t* page, size_t size ) {
  block_t* block = page->free;                    // page-local free list
```

[1]Do not confuse the word *page* with OS pages. A mimalloc page is larger and corresponds more closely to a *superblock* (Berger et al., 2000) or *subslab* (Liétar et al., 2019) in other allocators.

```
    if (block==NULL) return malloc_generic(size);    // slow path
    page->free = block->next;
    page->used++;
    return block;
  }
```



where

```
  struct block_t { struct block_t* next; }
```

There is just a single conditional and a pop in the fast path now. The `used` increment is needed to be able to efficiently determine when all objects in a page are freed. Many allocators use a *reap* design where a bump pointer is used initially when the page is empty (Berger et al., 2002; Feng and Berger, 2005; Liétar et al., 2019). We tested a variant of mimalloc with bump pointer allocation but across our benchmarks it was consistently about 2% worse. One reason might be that adding bump pointer allocation means there are now 2 conditionals in the fast path: either use the bump pointer or use the free list. Moreover, these conditionals cannot be predicted well as each one depends on the page where one happens to allocate in. Moreover, for security reasons it is not good to allocate predictably in a sequential way which rules out bump pointers too. As shown in Section 3.5, we initialize the free list in a fresh page in a randomized way.

2.3. The Local Free List

For the Koka and Lean runtimes, we wanted to bound the worst-case allocation and free times. In particular, when freeing large data structures, the number of recursive free calls need to be limited. Koka and Lean use reference counting in the runtime (similar to Swift and Python), but the problem occurs in any language with large data structures. Limiting the number of free calls with reference counting can be done with a simple limit counter and pushing the remaining pointers on a *deferred decrement* list.

The question is when to free again from this deferred decrement list? Here cooperation from allocator is necessary since the best time to do this is when the allocator is under pressure and needs to find more free space. The mimalloc allocator calls a user defined `deferred_free` callback when that happens. This is called from the slow path in mimalloc in the `malloc_generic` routine exactly when the page local free list is empty. This nicely combines with the single conditional in the fast path. We will see that we reuse this technique again, and put any more expensive operations into the generic routine guarded by the single conditional.

However, this does not quite work yet as there is no guarantee that the generic routine is called regularly: a user may free and allocate repeatedly within one page with the free list in the page never becoming empty. What we want instead is to ensure the generic routine is called after some fixed number of allocations.

Therefore, we shard the free list once more: we add a sharded *local free list* to each page and while we allocate from the regular free list, we put any freed objects on the *local* free list instead. This guarantees that the free list becomes empty after a fixed number of allocations. In the generic routine we can now simply move the local free list to the free list and keep allocating:

```
  page->free = page->local_free;  // move the list
  page->local_free = NULL;        // and the local list is empty again
```

Note again that we did not need to add a conditional in the fast path for this situation and put the work into the slow path. Now that `deferred_free` is guaranteed to be called regularly after a bounded number of allocations, we can also use it as a deterministic *heartbeat*. This is used in Lean as a form of portable timer to time-out threads if they take too long (for proofs). In that case

we cannot use wall-clock time since that would not be deterministic across machines while the heartbeat is.

2.4. The Thread Free List

In mimalloc, pages belong to a thread-local heap and allocation is always done in the local heap. This way no locks are needed for thread local allocations. Nevertheless, any thread can free an object. To avoid locks for thread local frees as well, we shard the free list one more final time and add a sharded *thread free list* per page, where other threads push freed objects in that page.

If a non-local free happens, we use atomic operations to push the freed object `p` atomically on the thread free list:

```
atomic_push( &page->thread_free, p );
```

where

```
void atomic_push( block_t** list, block_t* block ) {
  do { block->next = *list; }
  while (!atomic_compare_and_swap(list, block, block->next));
}
```

The beauty of the sharded thread free list is that it also reduces contention among threads since threads freeing in different pages do not contend with each other. On current architectures, uncontended atomic operations are very efficient and usually implemented as part of the cache consistency protocol (Schweizer et al., 2015).

Again, we use the generic routine to collect the thread free list and add it to the free list, just as we did with the local free list:

```
tfree = atomic_swap( &page->thread_free, NULL );
append( page->free, tfree );
```

Since the entire thread free list is moved at once, this effectively batches non-local free calls as well. This is especially important for asymmetric concurrent work loads where some threads predominantly free objects and others predominantly allocate. The snmalloc allocator (Liétar et al., 2019) was especially created to handle this situation well and also uses a batching technique to reduce expensive synchronization. This workload is tested by the xmalloc-testN benchmark in Section 4.

3. IMPLEMENTATION

Given the sharded free lists, we can now understand the full design of the allocator, where Figure 1 shows a detailed overview of the layout of the heap. Except for the sharded lists, the overall design is otherwise quite similar to other size-segregated thread-caching allocators.

3.1. Malloc

To allocate an object, mimalloc first gets a pointer to the thread local heap (`tlb`). From there it needs to find a page of the right size class. For small objects under 1Kb the heap contains a direct array of pointers to the first available page in that size class. For small object allocation, the code becomes:

```
void* malloc_small( size_t n ) {                    // 0 < n <= 1024
  heap_t*  heap  = tlb;
  page_t*  page  = heap->pages_direct[(n+7)>>3];  // divide up by 8
  block_t* block = page->free;
  if (block==NULL) return malloc_generic(heap,n); // slow path
  page->free = block->next;
```

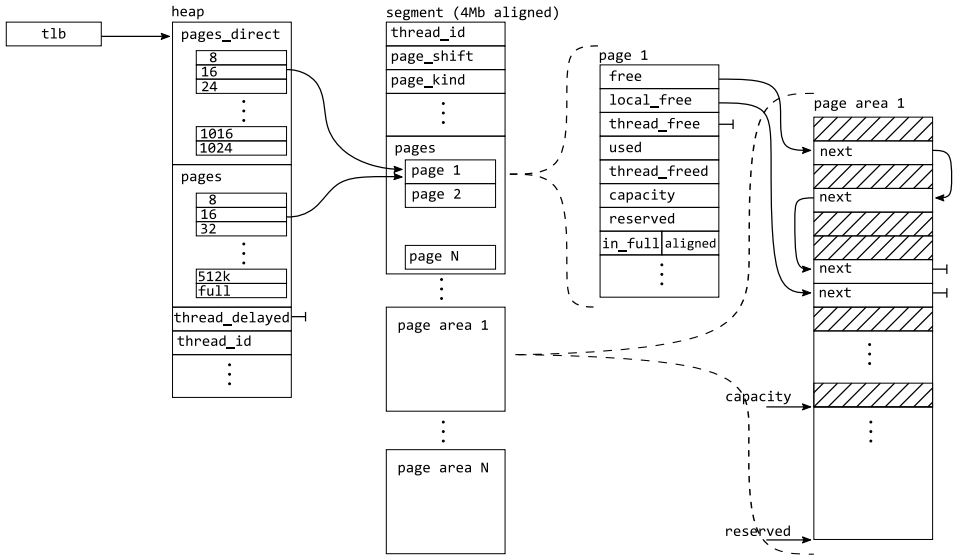


Fig. 1. Heap layout

```
  page->used++;
  return block;
}
```

which expands to efficient assembly with only one conditional.

As seen in Figure 1, the pages and the page meta-data all live in large *segments* (sometimes called *slab* or *arena* in other allocators). These segments are 4MiB (or larger for huge objects that are over 512KiB), and start with the segment- and page meta data, followed by the actual pages where the first page is shortened by the size of the meta data plus a guard page. There are three page sizes: for small objects under 8KiB the pages are 64KiB and there are 64 in a segment; for large objects under 512KiB there is one page that spans the whole segment, and finally huge objects over 512KiB have one page of the required size. The reason to still use a segment and single page for large and huge objects is to simplify the data structures and reduce the code size and complexity by having a consistent interface and code with few special cases. This pays off in practice and the code size of mimalloc is far smaller than most other allocators.

3.2. Free

Pages and their meta data are allocated in a segment mostly to reduce expensive allocation calls to the underlying OS, but there is another important reason: when freeing a pointer, we need to be able to find the page meta data belonging to that pointer. The way this is done in mimalloc is to align the segments to a 4MiB boundary. Finding the segment holding a pointer `p` can then be done by masking the lower bits. The code for `free` becomes:

```
void free( void* p ) {
  segment_t* segment = (segment_t*)((uintptr_t)p & ~(4*MB));
  if (segment==NULL) return;
  page_t*    page    = &segment->pages[(p - segment) >> segment->page_shift];
  block_t*   block   = (block_t*)p;
```

```
    if (thread_id() == segment->thread_id) {  // local free
      block->next = page->local_free;
      page->local_free = block;
      page->used--;
      if (page->used - page->thread_freed == 0) page_free(page);
    }
    else {                                    // non-local free
      atomic_push( &page->thread_free, block);
      atomic_incr( &page->thread_freed );
    }
  }
```

The free function first gets the segment pointer by masking the lower bits of the freed pointer `p`. When the pointer is `NULL`, the segment will be `NULL` too and we check for that. In the generated assembly this removes an explicit comparison operation as the bitwise *and* sets the zero-flag if the result is zero. From there we can calculate the page index by taking the difference and shifting by the segment `page_shift`: for small pages this is 16 (= 64KiB), while for large and huge pages it is 22 (= 4MiB) such that the index is always zero in those cases (as there is just one page). Again, by using a uniform representation we avoid special cases and conditionals in the fast path.

The main conditional tests whether this is a thread local free, or a free by another thread. Here mimalloc relies on an efficient `thread_id()` call to get the id of the current thread and comparing that to the `thread_id` field of the segment. On most operating systems this can be done very efficiently by loading the thread id from a fixed address of the thread local data (for example, on Linux on 64-bit Intel/AMD chips this at offset 0 relative to the `fs` register).

If the free is done by another thread, the object is pushed atomically on the `thread_free` list. Otherwise, the free is local and we simply push the object on the `local_free` list. We also test here if all objects are freed in that page and free the page in that case. We could skip this and instead only collect full free pages when looking for a fresh page in the slow path, but for certain work loads it turns out to be more efficient to try to make such pages available as early as possible.

Note that we read the thread shared `thread_freed` count without a read-barrier meaning there is tiny probability that we miss that all objects in the page were just all freed. However, that is okay – since we are guaranteed to call the generic allocation routine sometimes, we can collect any such pages later on (and indeed – on asymmetric workloads where some threads only allocate and others only free, the collection in the generic routine is the *only* way such pages get freed).

3.3. Generic Allocation

The generic allocation routine, `malloc_generic`, is our "slow path" which is guaranteed to be called every once in a while. This routine gives us the opportunity to do more expensive operations whose cost is amortized over many allocations, and can almost be seen as a form of garbage collector. In pseudo code:

```
void* malloc_generic( heap_t* heap, size_t size ) {
  deferred_free();
  foreach( page in heap->pages[size_class(size)] ) {
    page_collect(page);
    if (page->used - page->thread_freed == 0) {
      page_free(page);
    }
    else if (page->free != NULL) {
```

```
      return malloc(size);
    }
  }
  .... // allocate a fresh page and malloc from there
}

void page_collect(page) {
  page->free = page->local_free;  // move the local free list
  page->local_free = NULL;
  ... // move the thread free list atomically
}
```

The generic routine linearly walks through the pages of a size class and frees any pages that contain no more objects. It stops when it finds the first page that has free objects. In the actual implementation not all pages are immediately freed but some are retained a bit in a cache for possible future use; also, the maximum number of freed pages is bounded to limit the worst-case allocation time. When a page is found with free space, the page list is also rotated at that point so that a next search starts from that point.

3.4. The Full List

The implementation as described already performs very well on almost all of our wide range of benchmarks – except some. In particular, on the SpecMark gcc benchmark we observed a 30% slowdown compared to some other allocators. This anecdote shows that there is no silver bullet and an industrial strength memory allocator needs to address many corner cases that might show up only for particular workloads.

In the case of the gcc benchmark it happens to use its own custom allocators and allocate many large objects initially that than stay live for a long time. For mimalloc this leads to many (over 18000) *full* pages that are now traversed linearly every time in the generic allocation routine.

The solution that we implemented is to have a separate *full list* that holds all the pages that are full, and move those back to the regular page lists when an object is freed in such page. This fixes the gcc benchmark but unfortunately this seemingly small change introduces significant complexity for the multi-threaded case.

In particular, on a non-local free of an object in a full page, we need to somehow signal the owning heap that the page is no longer full, and if possible without taking an expensive lock. We are going to do this through a heap-owned list of *thread delayed free* blocks. In the generic routine, we first atomically take over this list and free all the blocks in the delayed free list normally – possibly moving pages from the full list back to the regular lists.

But how does a non-local free know whether to push on the page local *thread free* list, or whether do push on the owning heap *thread delayed free* list? For this we use the 2 least significant bits in the thread free list pointer to atomically encode 3 states: `NORMAL`, `DELAYED`, and `DELAYING`. Usually, the state is `NORMAL` and we push on the local thread free list. When a page is moved to the full list, we set the `DELAYED` state – signifying that non-local free operations should push on the owning heap delayed free list. While doing that, the `DELAYING` state is temporarily set to ensure the owning heap structure itself stays valid in case the owning thread terminates in the mean time. After a delayed free, the state is always set to `NORMAL` again since we only need one delayed free per page to check if the page is no longer full. This turns out to be an important optimization: again, with asymmetric concurrent workloads the freeing thread may free many objects and we should

ensure the more expensive initial delayed free is only done once. Without this optimization, the xmalloc-test benchmark is 30% slower.

3.5. Security

The design of mimalloc lends itself well to implement various security mitigations that one would consider required in browser environments for example. For a good overview we refer to Novark and Berger (2010) and Berger and Zorn (2006). We implemented a secure variant of mimalloc (called smimalloc) that implements various security mitigations:

- It puts OS guard pages in-between every mimalloc page such that heap overflow attacks are always limited to one mimalloc page and can never overflow into the heap meta data.
- The initial free list in a page is initialized randomly such that there is no predictable allocation pattern (to protect against *heap feng shui* (Sotirov, 2007)). Also, on a full list, the secure allocator will sometimes extend instead of using the local free list to increase randomness further.
- To guard against heap block-overflow attacks that overwrite the free list, we *xor*-encode the free list in each page. This prevents overwriting with known values but also allows efficient detection of such attack.
- Already, mimalloc efficiently supports multiple heaps. This can further increase security by allocating internal objects like virtual method tables etc. in a separate heap from other application allocated objects.

As we see in Section 4, the secure version of mimalloc is on average about 3% slower plain mimalloc. This was quite surprising to us as we initially expected much larger slowdowns due to the above mitigations.

4. EVALUATION

We tested mimalloc against many other top allocators over a wide range of benchmarks, ranging from various real world programs to synthetic benchmarks that see how the allocator behaves under more extreme circumstances. The benchmark suite is fully scripted and available on Github (Leijen, 2019a).

Allocators are interesting as there exists no algorithm that is generally optimal – for a given allocator one can usually construct a workload where it does not do so well. The goal is thus to find an allocation strategy that performs well over a wide range of benchmarks without suffering from underperformance in less common situations (which is what the second half of our benchmark set tests for).

In our benchmarks, mimalloc always outperforms all other leading allocators (jemalloc, tcmalloc, Hoard, etc), and usually uses less memory (up to 25% more in the worst case). A nice property is that it does *consistently* well over the wide range of benchmarks: only snmalloc shares this property while all other allocators exhibit sudden (severe) underperformance in certain situations. We try to highlight and explain these situations in the text and hope these insights can lead to improvements in other allocator designs as well.

4.1. Allocators

We tested mimalloc with 9 leading allocators over 12 benchmarks and the SpecMark benchmarks. The tested allocators are:

- **mi**: The mimalloc allocator (Leijen, 2019b), using version tag `v1.0.0`. We also test a secure version of mimalloc as **smi** which uses the techniques described in Section 3.5.

- **tc**: The tcmalloc allocator (Google, 2014) which comes as part of the Google performance tools and is used in the Chrome browser. Installed as package `libgoogle-perftools-dev` version `2.5-2.2ubuntu3`.
- **je**: The jemalloc allocator by Evans (2006) is developed at Facebook and widely used in practice, for example in FreeBSD and Firefox. Using version tag 5.2.0.
- **sn**: The snmalloc allocator is a recent concurrent message passing allocator by Liétar et al. (2019). Using `git-0b64536b`.
- **rp**: The rpmalloc allocator uses 32-byte aligned allocations and is developed by Jansson (2017) at Rampant Pixels. Using version tag 1.3.1.
- **hd**: The Hoard allocator by Berger et al. (2000). This is one of the first multi-thread scalable allocators. Using version tag 3.13.
- **glibc**: The system allocator. Here we use the glibc allocator (which is originally based on Ptmalloc2), using version 2.27.0. Note that version 2.26 significantly improved scalability over earlier versions.
- **sm**: The Supermalloc allocator by Kuszmaul (2015) uses hardware transactional memory to speed up parallel operations. Using version `git-709663fb`.
- **tbb**: The Intel TBB allocator that comes with the Thread Building Blocks (TBB) library (Intel, 2017; Kukanov and Voss, 2007; Hudson et al., 2006). Installed as package `libtbb-dev`, version `2017~U7-8`.

All allocators run exactly the same benchmark programs on Ubuntu 18.04.1 and use `LD_PRELOAD` to override the default allocator. The wall-clock elapsed time and peak resident memory (rss) are measured with the `time` program. The average scores over 5 runs are used. Performance is reported relative to mimalloc, e.g. a time of 1.5× means that the program took 1.5× longer than mimalloc.

4.2. Benchmarks

The first set of benchmarks are real world programs and consist of:

- **cfrac**: by Dave Barrett, implementation of continued fraction factorization which uses many small short-lived allocations – exactly the workload we are targeting for Koka and Lean.
- **espresso**: a programmable logic array analyzer, described by Grunwald, Zorn, and Henderson (1993b) in the context of cache aware memory allocation.
- **barnes**: a hierarchical n-body particle solver (Barnes and Hut, 1986) which uses relatively few allocations compared to `cfrac` and `espresso`. Simulates the gravitational forces between 163840 particles.
- **leanN**: The Lean compiler by de Moura et al (2015), version 3.4.1, compiling its own standard library concurrently using N threads (`./lean --make -j N`). Big real-world workload with intensive allocation.
- **redis**: running the redis 5.0.3 server on 1 million requests pushing 10 new list elements and then requesting the head 10 elements. Measures the requests handled per second.
- **larsonN**: by Larson and Krishnan (1998). Simulates a server workload using 100 separate threads which each allocate and free many objects but leave some objects to be freed by other threads. Larson and Krishnan observe this behavior (which they call bleeding) in actual server applications, and the benchmark simulates this.

The second set of benchmarks are stress tests and consist of:

- **alloc-test**: a modern allocator test developed by OLogN Technologies AG (ITHare.com) (2018). Simulates intensive allocation workloads with a Pareto size distribution. The alloc-testN benchmark runs on N cores doing $100 \cdot 10^6$ allocations per thread with objects up to 1KiB in size. Using commit `94f6cb` (master, 2018-07-04)

- **sh6bench**: by MicroQuill (2006) as part of SmartHeap. Stress test where some of the objects are freed in a usual last-allocated, first-freed (LIFO) order, but others are freed in reverse order. Using the public source (retrieved 2019-01-02)
- **sh8benchN**: by MicroQuill (2006) as part of SmartHeap. Stress test for multi-threaded allocation (with N threads) where, just as in larson, some objects are freed by other threads, and some objects freed in reverse (as in sh6bench). Using the public source (retrieved 2019-01-02)
- **xmalloc-testN**: by Lever and Boreham (2000) and Christian Eder. We use the updated version from the SuperMalloc repository (Kuszmaul, 2015). This is a more extreme version of the larson benchmark with 100 purely allocating threads, and 100 purely deallocating threads with objects of various sizes migrating between them. This asymmetric producer/consumer pattern is usually difficult to handle by allocators with thread-local caches.
- **cache-scratch**: by Berger et al. (2000). Introduced with the Hoard allocator to test for passive-false sharing of cache lines: first some small objects are allocated and given to each thread; the threads free that object and allocate immediately another one, and access that repeatedly. If an allocator allocates objects from different threads close to each other this will lead to cache-line contention.

4.3. On a 16-core AMD EPYC

Figure 2 (and 6 for memory in the Appendix) shows the benchmark results on a r5a.4xlarge (Amazon EC2, 2019) instance consisting of a 16-core AMD EPYC 7000 at 2.5GHz with 128GB ECC memory, running Ubuntu 18.04.1 with LibC 2.27 and GCC 7.3.0. We excluded SuperMalloc here as it use transactional memory instructions that are usually not supported in a virtualized environment.

In the first five benchmarks we can see mimalloc outperforms the other allocators moderately, but we also see that all these modern allocators perform well – the times of large performance differences in regular workloads are over. In cfrac and espresso, mimalloc is a tad faster than tcmalloc and jemalloc, but a solid 10% faster than all other allocators on espresso. The tbb allocator does not do so well here and lags more than 20% behind mimalloc. The cfrac and espresso programs do not use much memory (~1.5MB) so it does not matter too much, but still mimalloc uses about half the resident memory of tcmalloc.

The leanN program is most interesting as a large realistic and concurrent workload and there is a 8% speedup over tcmalloc. This is quite significant: if Lean spends 20% of its time in the allocator that means that mimalloc is 1.3× faster than tcmalloc here. This is surprising as that is *not* measured in a pure allocation benchmark like alloc-test. We conjecture that we see this outsized improvement here because mimalloc has better locality in the allocation which improves performance for the *other* computations in a program as well.

The redis benchmark shows more differences between the allocators where mimalloc is 14% faster than jemalloc. On this benchmark tbb (and Hoard) do not do well and are over 40% slower.

The larson server workload which allocates and frees objects between many threads shows even larger differences, where mimalloc is more than 2.5× faster than tcmalloc and jemalloc which is quite surprising for these battle tested allocators – probably due to the object migration between different threads. This is a difficult benchmark for other allocators too where mimalloc is still 48% faster than the next fastest (snmalloc).

The second benchmark set tests specific aspects of the allocators and shows even more extreme differences between them.

The alloc-test is very allocation intensive doing millions of allocations in various size classes. The test is scaled such that when an allocator performs almost identically on alloc-test1 as alloc-testN it means that it scales linearly. Here, tcmalloc, snmalloc, and Hoard seem to scale less well and do

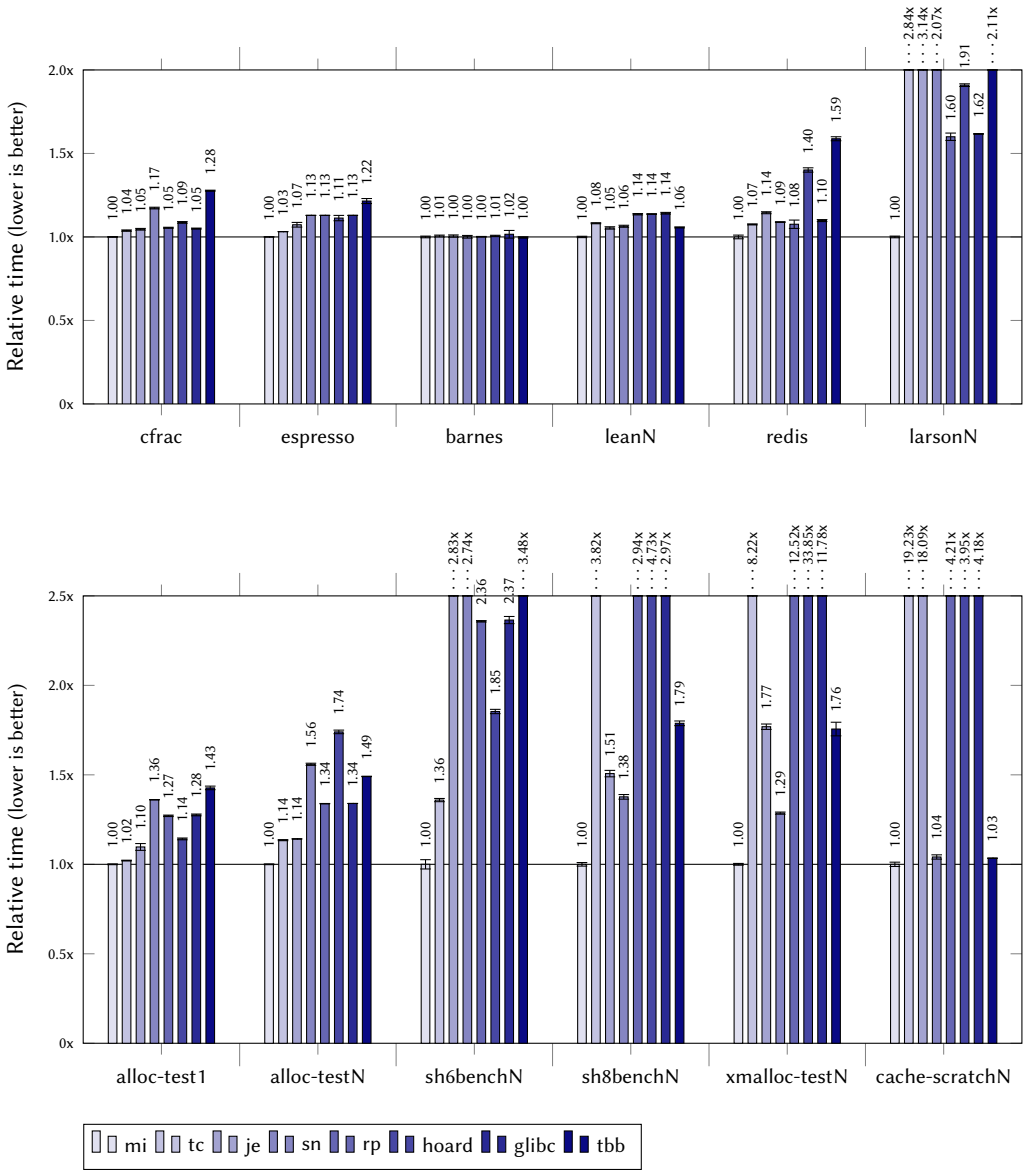


Fig. 2. Time benchmark on a 16-core AMD Epyc r5a-4xlarge instance. Benchmarks ending with "N" run in parallel on all cores.

more than 10% worse on the multi-core version. Even the best allocators (tcmalloc and jemalloc) are more than 10% slower as mimalloc here.

Also in sh6bench mimalloc does much better than the others (more than 2× faster than jemalloc). We cannot explain this well but believe it is caused in part by the “reverse” free-ing pattern in sh6bench.

Again in sh8bench the mimalloc allocator handles object migration between threads much better and is over 36% faster than the next best allocator, snmalloc. Whereas tcmalloc did well on sh6bench, the addition of object migration caused it to be almost 3 times slower than before.

The xmalloc-testN benchmark simulates an asymmetric workload where some threads only allocate, and others only free. The snmalloc allocator was especially developed to handle this case well as it often occurs in concurrent message passing systems. Here we see that the mimalloc technique of having non-contended sharded thread free lists pays off and it even outperforms snmalloc. Only jemalloc also handles this reasonably well, while the others underperform by a large margin. The optimization on mimalloc to do a *delayed free* only once for full pages is quite important – without it mimalloc is almost twice as slow (as then all frees contend again on the single heap delayed free list).

The cache-scratch benchmark also demonstrates the different architectures of the allocators nicely. With a single thread they all perform the same, but when running with multiple threads the allocator induced false sharing of the cache lines causes large run-time differences, where mimalloc is more than 18× faster than jemalloc and tcmalloc! Crundal (2016) describes in detail why the false cache line sharing occurs in the tcmalloc design, and also discusses how this can be avoided with some small implementation changes. Only snmalloc and tbb also avoid the cache line sharing like mimalloc. Kukanov and Voss (2007) describe in detail how the design of tbb avoids the false cache line sharing.

4.4. On a 4-core Intel Xeon workstation

Figure 3 and 7 show the benchmark results on an HP Z4-G4 workstation with a 4-core Intel® Xeon® W2123 at 3.6 GHz with 16GB ECC memory, running Ubuntu 18.04.1 with LibC 2.27 and GCC 7.3.0. This time SuperMalloc (sm) is included as this platform supports hardware transactional memory. Unfortunately, there are no entries for SuperMalloc in the leanN and xmalloc-testN benchmarks as it faulted on those. We also add the secure version of mimalloc as **smi**.

Overall, the relative results are quite similar as before. Most allocators fare better on the larsonN benchmark now – either due to architectural changes (AMD vs. Intel) or because there is just less concurrency. Unfortunately, the SuperMalloc faulted on the leanN and xmalloc-testN benchmarks.

The secure mimalloc version uses guard pages around each (mimalloc) page, encodes the free lists and uses randomized initial free lists, and we expected it would perform quite a bit worse – but on the first benchmark set it performed only about 3% slower on average, and is second best overall.

4.5. SpecMark 2019

We also ran SpecMark 2019 benchmarks. Most benchmarks there do not allocate a lot and all the modern allocators perform mostly identical for most of them. There are only 4 of them that show larger differences, which we show in Figure 4 and 5: 602.gcc, 620.omnetpp, 623.xalancbmk, and 648.exchange2.

On these benchmarks mimalloc does well but is slightly slower than tcmalloc, jemalloc, and snmalloc, on omnetpp and xalancbmk. As discussed in Section 3.4, the gcc benchmarks allocates a lot of initial long lived data and we needed the *full* list to avoid long searches. We conjecture this is happening in tcmalloc and tbb as well, as both have a similar underperformance of about 30% (just like mimalloc before the optimization). We see something similar happen in the xalancbmk benchmark for rp and glibc but we are not sure what is the cause of that.

In Figure 5 the relative peak memory usage is shown. Interestingly, the gcc benchmark shows two outliers too, but this time Hoard and tbb underperform by 30%. On the exchange2 benchmark it is surprising to see that both tcmalloc and jemalloc use significantly more memory than mimalloc

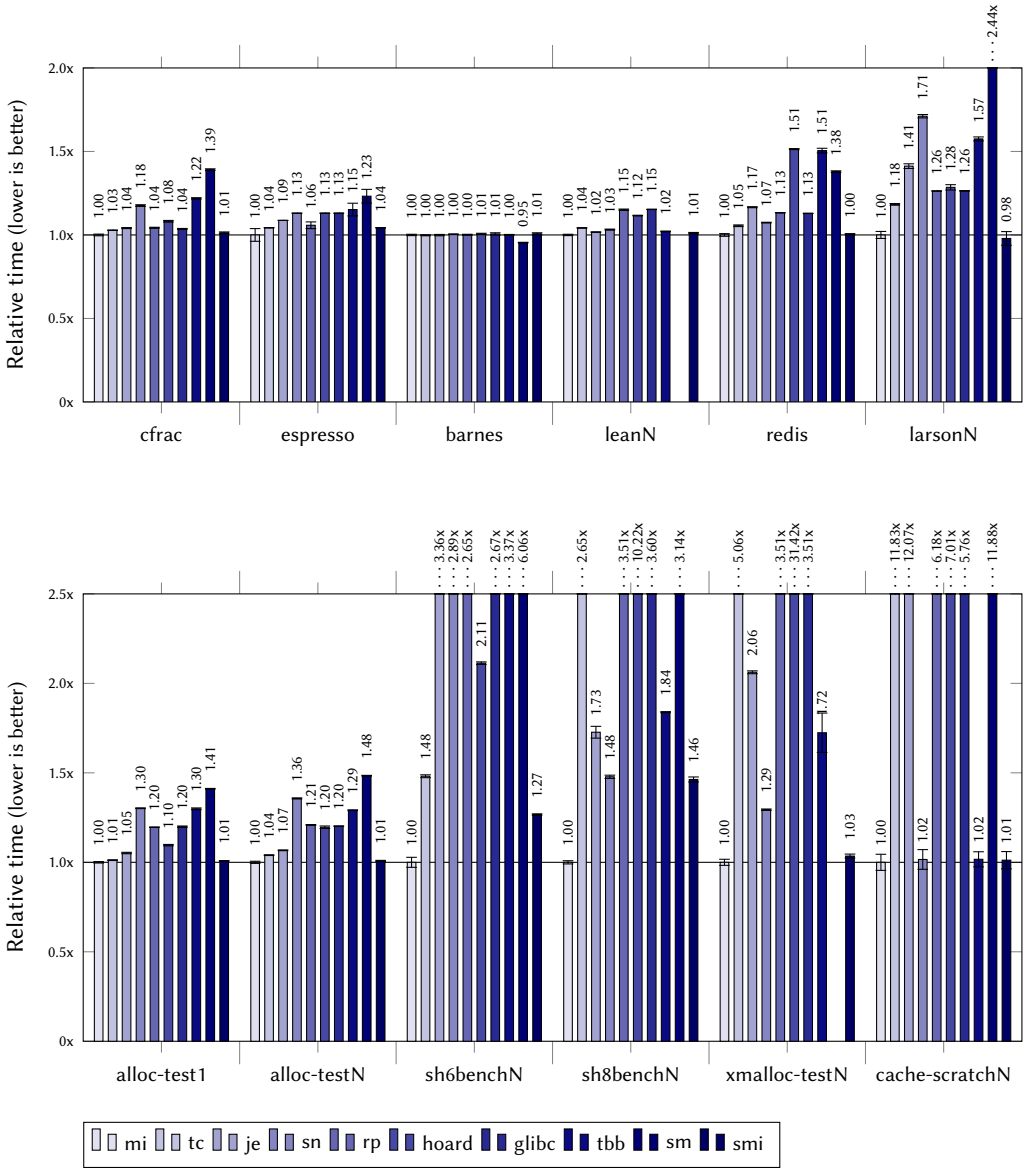


Fig. 3. Time benchmark on a 4-core Intel Xeon workstation. Benchmarks ending with "N" run in parallel on all cores.

even though especially jemalloc is optimized to reduce the resident memory usage for long running server programs.

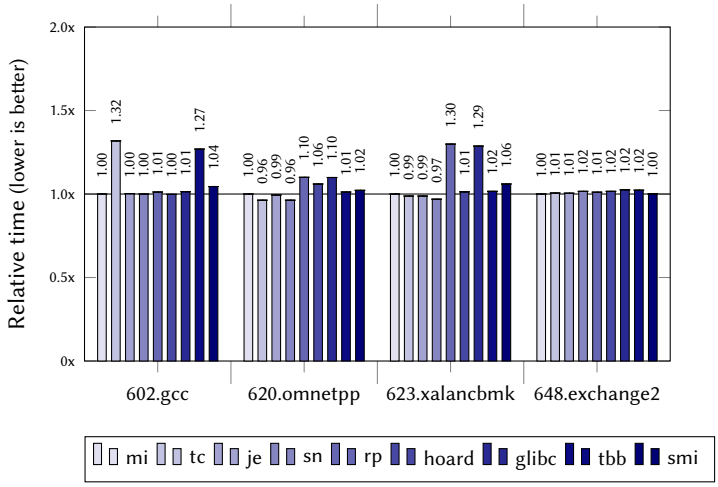


Fig. 4. Time benchmark on a 4-core Intel Xeon workstation for selected SpecMark 2019 benchmarks.

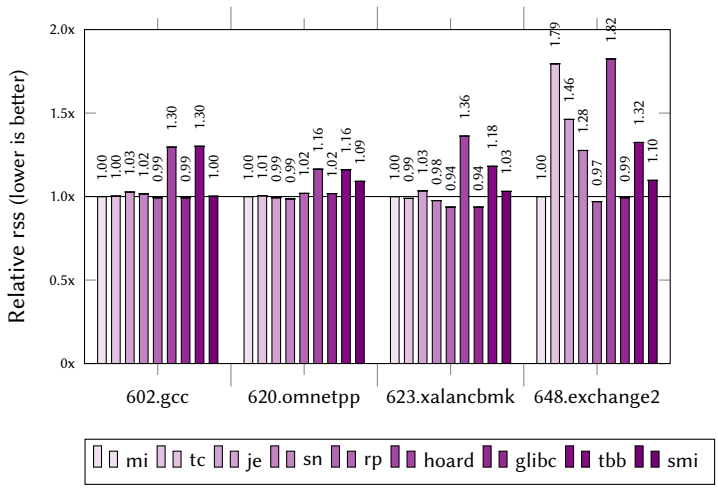


Fig. 5. Peak memory usage on a 4-core Intel Xeon workstation for selected SpecMark 2019 benchmarks.

5. RELATED WORK

Feng and Berger's VAM (Feng and Berger, 2005) is the allocator design most closely related to mimalloc. VAM pioneered the idea of prioritizing application reference locality over reducing memory fragmentation and our sharded free list design improves on VAM's original design. VAM maintained free lists per 4k hardware page and supported bump-pointer allocations (which we considered but rejected). As many allocators contemporaneous with VAM did, VAM treated large

and medium-sized objects differently than small objects by incorporating inline meta data with each object to support a best-fit allocation strategy. VAM was not a multi-threaded allocator design, as mimalloc is, and its implementation is not currently available for measuring.

Grunwald et al. (1993a) highlight the impact of allocator design on overall application reference locality and argue that a segregated size-class approach, as implemented in QuickFit (Weinstock and Wulf, 1988) would provide better reference locality. While Grunwald argues that QuickFit is only part of a more general allocator solution, unlike Grunwald or VAM, mimalloc demonstrates that a uniform approach to object representation across all sizes leads to significant benefits in reduced complexity and improved performance.

The Intel TBB (Thread Building Block) multi-threaded allocator (Kukanov and Voss, 2007; Hudson et al., 2006) has elements in common with mimalloc. It uses size-segregated bins, has thread-local free lists, allocates from a private free list and and has a public free list per bin that foreign threads return local objects to. Unlike mimalloc, TBB does not have a separate private free list that local objects are returned on, choosing instead to immediately reuse freed objects instead of deferring reuse. As our results show, the mimalloc allocate/free fast past is significantly faster than TBB (e.g., 50% faster in the redis benchmark) and mimalloc also scales better than TBB in the multi-threaded benchmarks on both 4 and 16-core systems.

snmalloc is a recently published allocator that focuses on improving the performance of multi-threaded producer/consumer workloads (Liétar et al., 2019), as exemplified by the xmalloc-testN benchmark. snmalloc supports high performance sharing of objects between threads, introducing a novel radix-tree structure to avoid potential bottlenecks with different consumer threads contending with each other on returning an object to the same producer. mimalloc handles contention between threads performing frees of non-local objects by sharding the thread free list in every page.

6. CONCLUSION

We present mimalloc, an allocator motivated by the need to support deferred reference decrements in language runtimes and focused on improving the overall reference locality of an application. mimalloc provides three sharded free lists per software page (64KiB), increasing overall locality, reducing multi-threaded contention, and supporting temporal cadence, where slow-path operations are deferred but guaranteed to happen with regularity. To avoid costly branches on the fast path, mimalloc simplifies object representation and eliminates complexities such as doing bump-pointer allocation, representing medium-objects differently, etc. Comparing against state-of-the-art commercial allocator implementations, we show that mimalloc consistently outperforms other allocators in their default configuration including on both single-threaded workloads, such as redis, as well as on multi-threaded stress tests. mimalloc is implemented in C, is freely available on github (Leijen, 2019b) and with its simple and small code base is particularly amenable to being integrated into other language runtimes.

Acknowledgements

We would like to thank Matthew Parkison, and the other authors of snmalloc, for the valuable feedback, and encouragement to include the xmalloc-testN benchmark.

REFERENCES

Amazon EC2. "Cloud Instance Types." 2019. `https://aws.amazon.com/ec2/instance-types/`.

Josh Barnes, and Piet Hut. "A Hierarchical O(N Log N) Force-Calculation Algorithm." *Nature* 324 (December): 446–449. Dec. 1986. doi:10.1038/324446a0.

Emery D. Berger, Kathryn S. McKinley, Robert D. Blumofe, and Paul R. Wilson. "Hoard: A Scalable Memory Allocator for Multithreaded Applications." In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, 117–128. ASPLOS IX. ACM, Cambridge, Massachusetts, USA. 2000. doi:10.1145/378993.379232.

Emery D. Berger, and Benjamin G. Zorn. “DieHard: Probabilistic Memory Safety for Unsafe Languages.” In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 158–168. PLDI ’06. Ottawa, Ontario, Canada. 2006. doi:10.1145/1133981.1134000.

Emery D Berger, Benjamin G Zorn, and Kathryn S McKinley. *Reconsidering Custom Memory Allocation*. Volume 37. 11. ACM. 2002.

Timothy Crundal. “Reducing Active-False Sharing in TCMalloc.” 2016. `http://courses.cecs.anu.edu.au/courses/CSPROJECTS/16S1/Reports/Timothy*Crundal*Report.pdf`. CS16S1 project at the Australian National University.

Jason Evans. “Jemalloc.” In *Proceedings of the 2006 BSDCan Conference*. BSDCan’06’. Ottowa,CA. May 2006. `http://people.freebsd.org/~jasone/jemalloc/bsdcan2006/jemalloc.pdf`.

Yi Feng, and Emery D. Berger. “A Locality-Improving Dynamic Memory Allocator.” In *Proceedings of the 2005 Workshop on Memory System Performance*, 68–77. Chicago, Illinois, USA. Jan. 2005. doi:10.1145/1111583.1111594.

Google. “Tcmalloc.” 2014. `https://github.com/gperftools/gperftools`.

Dirk Grunwald, Benjamin Zorn, and Robert Henderson. “Improving the Cache Locality of Memory Allocation.” In *ACM SIGPLAN Notices*, 28:177–186. 6. ACM. 1993.

Dirk Grunwald, Benjamin Zorn, and Robert Henderson. “Improving the Cache Locality of Memory Allocation.” *SIGPLAN Notices* 28 (6). ACM: 177–186. Jun. 1993. doi:10.1145/173262.155107.

Richard L Hudson, Bratin Saha, Ali-Reza Adl-Tabatabai, and Benjamin C Hertzberg. “McRT-Malloc: A Scalable Transactional Memory Allocator.” In *Proceedings of the 5th International Symposium on Memory Management*, 74–83. ACM. 2006.

Intel. “Thread Building Blocks (TBB).” 2017. `https://www.threadingbuildingblocks.org/`.

Mattias Jansson. “Rpmalloc.” 2017. `https://github.com/rampantpixels/rpmalloc`.

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

Bradley C. Kuszmaul. “SuperMalloc: A Super Fast Multithreaded Malloc for 64-Bit Machines.” In *Proceedings of the 2015 International Symposium on Memory Management*, 41–55. ISMM’15. ACM, Portland, OR, USA. 2015. doi:10.1145/2754169.2754178.

Per-Åke Larson, and Murali Krishnan. “Memory Allocation for Long-Running Server Applications.” In *Proceedings of the 1998 International Symposium on Memory Management*, 176–185. ISMM’98. 1998.

Daan Leijen. “Koka: Programming with Row Polymorphic Effect Types.” In *MSFP’14, 5th Workshop on Mathematically Structured Functional Programming*. 2014. doi:10.4204/EPTCS.153.8.

Daan Leijen. “Type Directed Compilation of Row-Typed Algebraic Effects.” In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL’17)*, 486–499. Paris, France. Jan. 2017. doi:10.1145/3009837.3009872.

Daan Leijen. “Mimalloc Repository.” Jun. 2019a. `https://github.com/microsoft/mimalloc`.

Daan Leijen. “Mimalloc Benchmark Repository.” Jun. 2019b. `https://github.com/daanx/mimalloc-bench`.

C. Lever, and D. Boreham. “Malloc() Performance in a Multithreaded Linux Environment.” In *USENIX Annual Technical Conference, Freenix Session*. San Diego, CA. Jun. 2000. malloc-test available from `https://github.com/kuszmaul/SuperMalloc/tree/master/tests`.

Paul Liétar, Theodore Butler, Sylvan Clebsch, Sophia Drossopoulou, Juliana Franco, Matthew J Parkinson, Alex Shamis, Christoph M Wintersteiger, and David Chisnall. “Snmalloc: A Message Passing Allocator.” In *Proceedings of the 2019 ACM SIGPLAN International Symposium on Memory Management*, 122–135. ACM. 2019.

Paul Liétar, Theodore Butler, Sylvan Clebsch, Sophia Drossopoulou, Juliana Franco, Matthew J. Parkinson, Alex Shamis, Christoph M. Wintersteiger, and David Chisnall. “Snmalloc: A Message Passing Allocator.” In *Proceedings of the 2019 International Symposium on Memory Management*. ISMM’19. Phoenix, AZ. 2019. `https://github.com/Microsoft/snmalloc`.

MicroQuill. “SmartHeap.” 2006. `http://www.microquill.com`. sh6bench available at `http://www.microquill.com/smartheap/shbench/bench.zip`, sh8benc available at `http://www.microquill.com/smartheap/SH8BENCH.zip`.

Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. “The Lean Theorem Prover (System Description).” In *Automated Deduction - CADE-25*, edited by Amy P. Felty and Aart Middeldorp, 378–388. Springer International Publishing. 2015.

Gene Novark, and Emery D. Berger. “DieHarder: Securing the Heap.” In *Proceedings of the 17th ACM Conference on Computer and Communications Security*, 573–584. CCS ’10. Chicago, Illinois, USA. 2010. doi:10.1145/1866307.1866371.

OLogN Technologies AG (ITHare.com). “Testing Memory Allocators: ptmalloc2 vs Tcmalloc vs Hoard vs Jemalloc, While Trying to Simulate Real-World Loads.” Jul. 2018. `http://ithare.com/testing-memory-allocators-ptmalloc2-tcmalloc-hoard-jemalloc-while-trying-to-simulate-real-world-loads/`. Test available at `https://github.com/node-dot-cpp/alloc-test`.

Michel F Sanner, and others. “Python: A Programming Language for Software Integration and Development.” *J Mol Graph Model* 17 (1): 57–61. 1999.

H. Schweizer, M. Besta, and T. Hoefler. “Evaluating the Cost of Atomic Operations on Modern Architectures.” In *2015 International Conference on Parallel Architecture and Compilation (PACT)*, 445–456. Oct. 2015. doi:10.1109/PACT.2015.24.

A. Sotirov. “Heap Feng Shui in JavaScript.” 2007. `https://www.blackhat.com/presentations/bh-europe-07/FSotirov/Presentation/bh-eu-07-sotirov-apr19.pdf`. Blackhat Europe.

Sebastian Ullrich, and Leonardo de Moura. “Counting Immutable Beans – Reference Counting Optimized for Purely Functional Programming.” In *Proceedings of the 31st Symposium on Implementation and Application of Functional Languages (IFL’19)*. Singapore. Sep. 2019.

Charles B Weinstock, and William A Wulf. “An Efficient Algorithm for Heap Storage Allocation.” *ACM Sigplan Notices* 23 (10). ACM: 141–148. 1988.

Michael Wilde, Mihael Hategan, Justin M Wozniak, Ben Clifford, Daniel S Katz, and Ian Foster. “Swift: A Language for Distributed Parallel Scripting.” *Parallel Computing* 37 (9). Elsevier: 633–652. 2011.

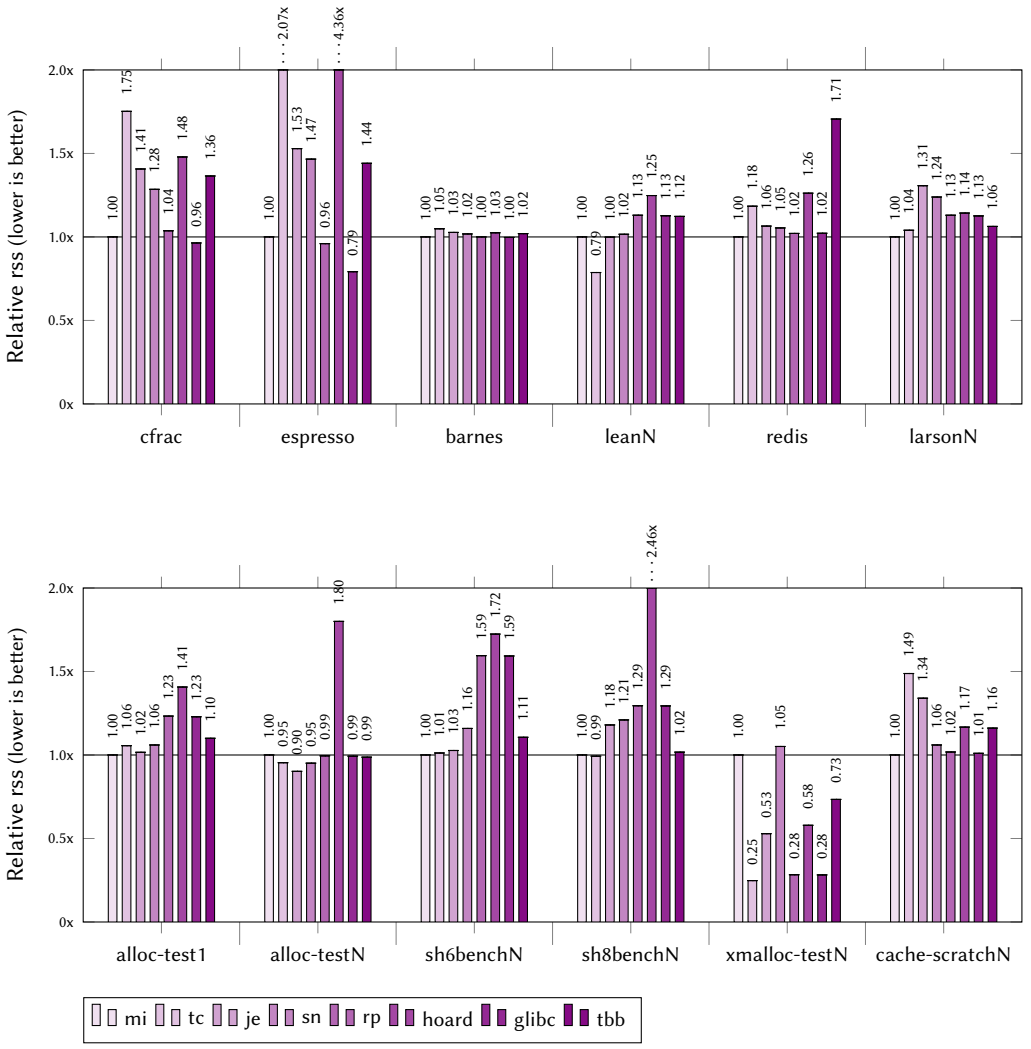


Fig. 6. Peak memory usage on a 16-core AMD Epyc r5a-4xlarge instance. (xmalloc-testN is not normalized and should be disregarded)

APPENDICES

A. EVALUATION OF PEAK WORKING MEMORY

Figures 6 and 7 show the peak working memory (RSS) relative to mimalloc. These figures correspond to the earlier performance figures 2 and 3 respectively. Note that the memory usage of xmalloc-testN should be disregarded as the faster the benchmark runs, the more memory it uses. Also the cfrac, espresso, and cache-scratchN benchmarks use just little active memory and the differences in RSS are not very important here.

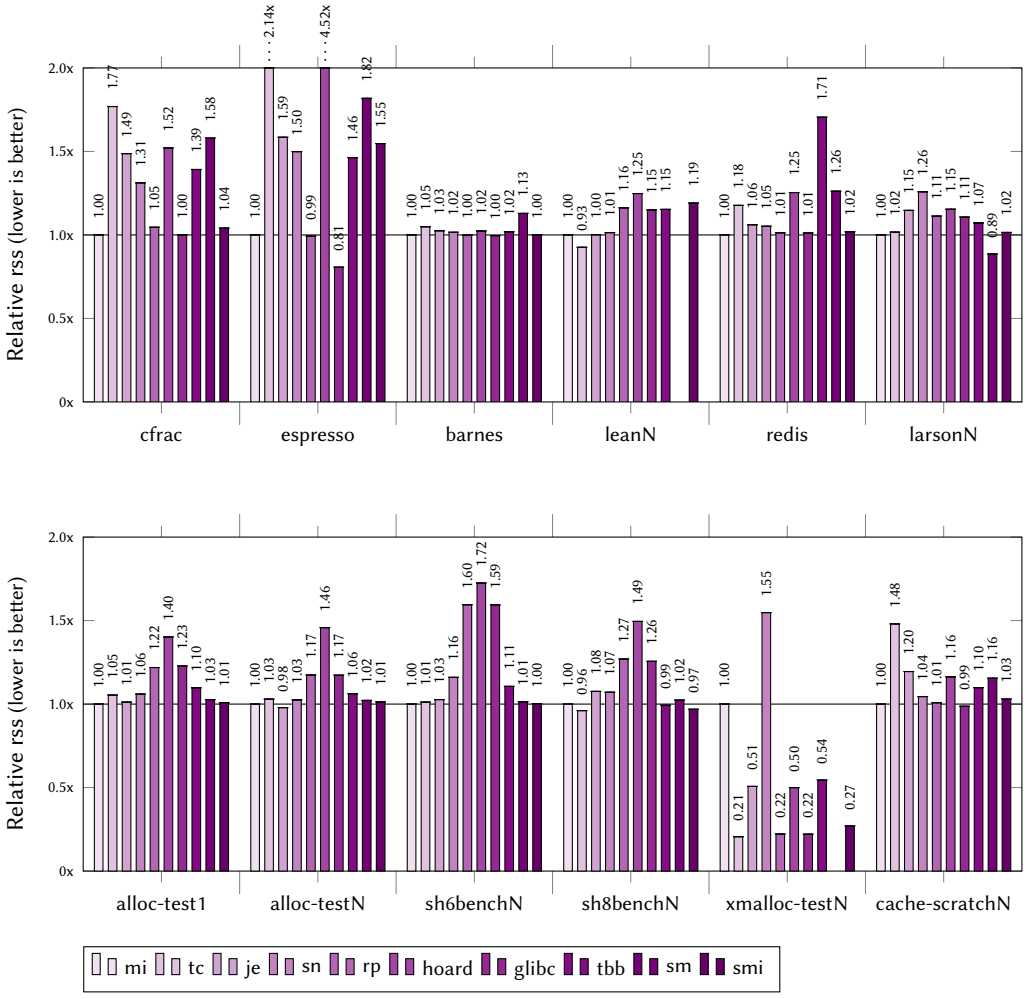


Fig. 7. Peak memory usage on a 4-core Intel Xeon workstation. (xmalloc-testN is not normalized and should be disregarded)

B. FREE LISTS

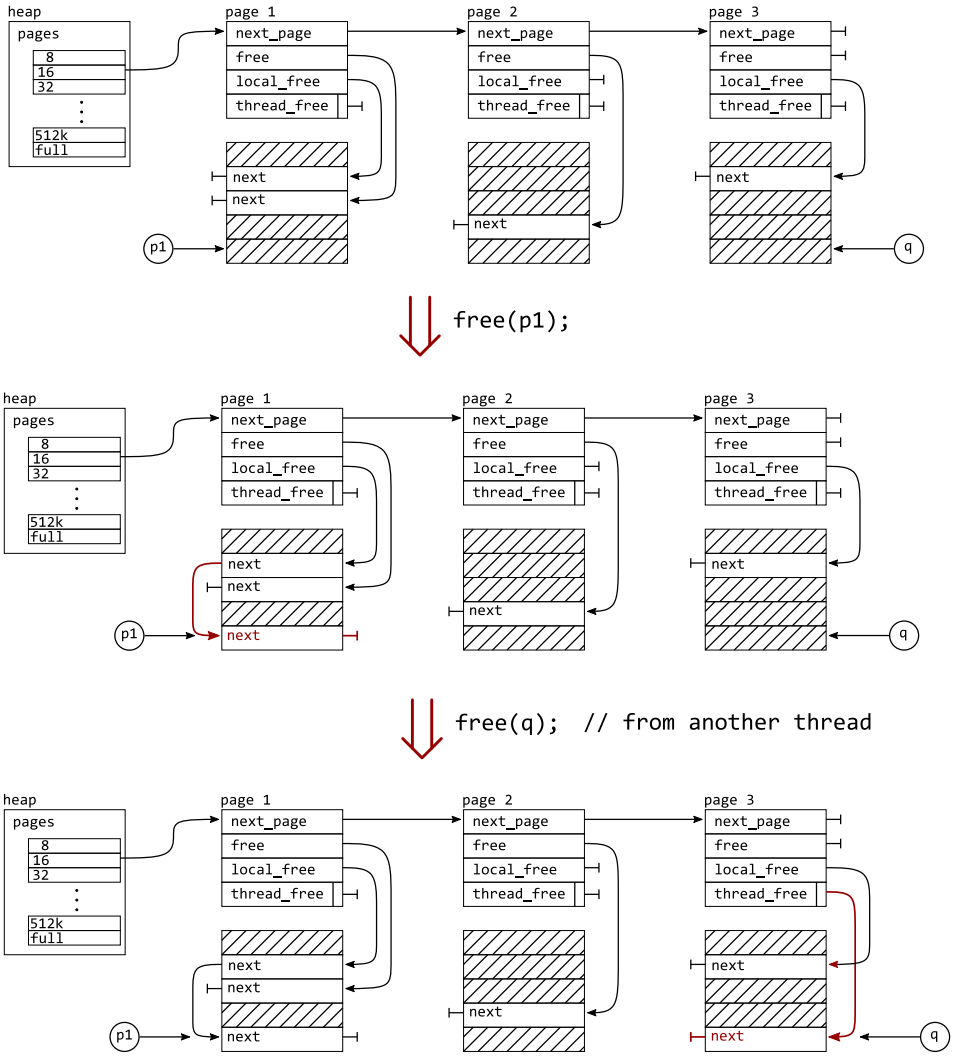


Fig. 8. Freeing objects

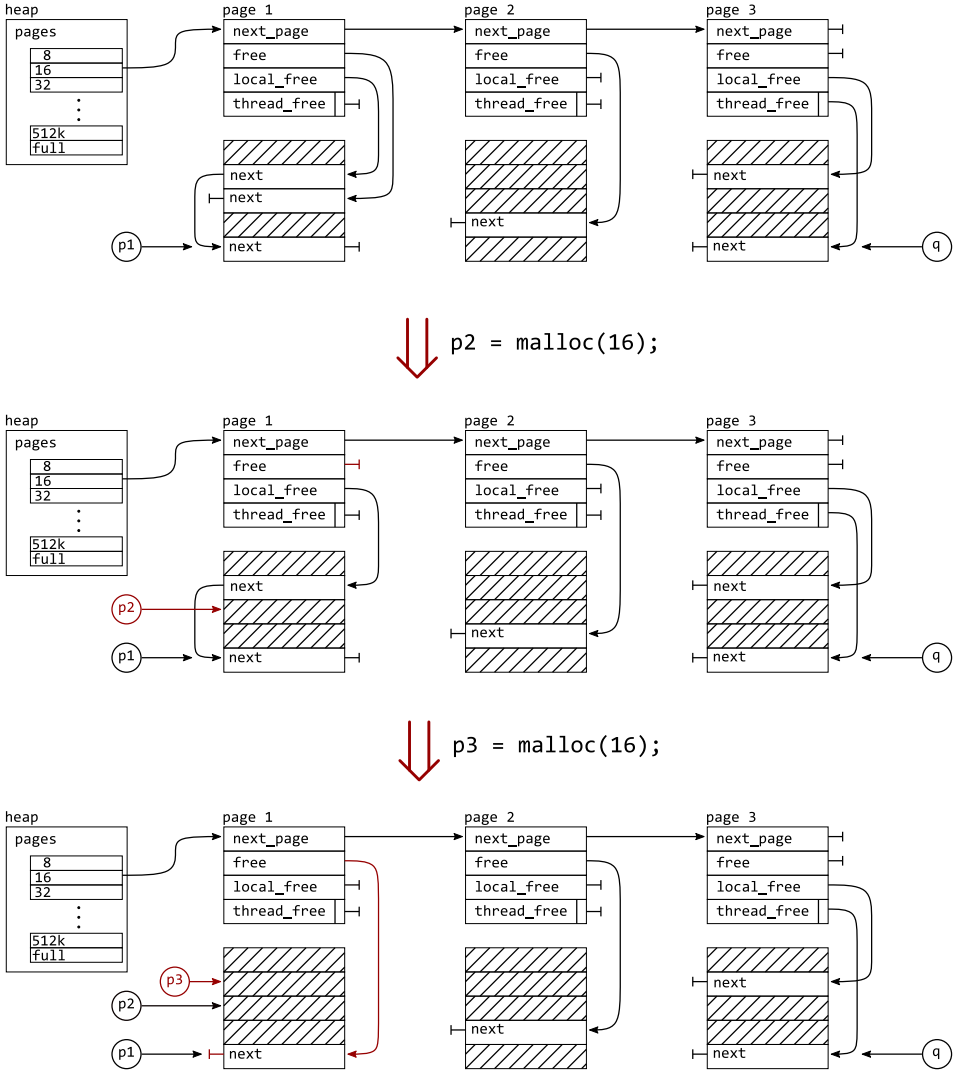


Fig. 9. Allocating objects