SuperMalloc: A Super Fast Multithreaded Malloc for 64-bit Machines

Bradley C. Kuszmaul
MIT CSAIL, Cambridge, MA, USA
bradley@mit.edu

Abstract

SuperMalloc is an implementation of `malloc(3)` originally designed for X86 Hardware Transactional Memory (HTM). It turns out that the same design decisions also make it fast even without HTM. For the malloc-test benchmark, which is one of the most difficult workloads for an allocator, with one thread SuperMalloc is about 2.1 times faster than the best of DLmalloc, JEmalloc, Hoard, and TBBmalloc; with 8 threads and HTM, SuperMalloc is 2.75 times faster; and on 32 threads without HTM SuperMalloc is 3.4 times faster. SuperMalloc generally compares favorably with the other allocators on speed, scalability, speed variance, memory footprint, and code size.

SuperMalloc achieves these performance advantages using less than half as much code as the alternatives. SuperMalloc exploits the fact that although physical memory is always precious, virtual address space on a 64-bit machine is relatively cheap. It allocates 2 MiB chunks which contain objects all the same size. To translate chunk numbers to chunk metadata, SuperMalloc uses a simple array (most of which is uncommitted to physical memory). SuperMalloc takes care to avoid associativity conflicts in the cache: most of the size classes are a prime number of cache lines, and nonaligned huge accesses are randomly aligned within a page. Objects are allocated from the fullest non-full page in the appropriate size class. For each size class, SuperMalloc employs a 10-object per-thread cache, a per-CPU cache that holds about a level-2-cache worth of objects per size class, and a global cache that is organized to allow the movement of many objects between a per-CPU cache and the global cache using $O(1)$ instructions. SuperMalloc prefetches everything it can before starting a critical section, which makes the critical sections run fast, and for HTM improves the odds that the transaction will commit.

Categories and Subject Descriptors D.4.2 Operating Systems [*Storage Management*] Allocation/deallocation strategies

Keywords Memory Allocation Library, malloc, Virtual Memory

1. Introduction

C/C++ dynamic memory allocation functions (`malloc(3)` and `free(3)`) can impact the cost of running applications. The cost can show up in several ways: allocation operations can be slow for serial programs, they can fail to scale with the number of cores in a multithreaded multicore environment, they can be occasionally slow, and they can induce a large memory footprint. Furthermore, if the allocator itself is too complex, it can inhibit improvements. We can divide these problems roughly into three categories: speed, footprint, and complexity.

The rest of this section explores this space in the context of several allocators. The allocators include DLmalloc [29]; Hoard [4]; JEmalloc [14]; TBBmalloc [27]; and our allocator, SuperMalloc. Figure 1 shows the code size for the allocators.

Many modern allocators take advantage of the fact that the operating system allocates memory in two steps corresponding to virtual allocation and physical allocation: First a system call such as `mmap()` allocates a contiguous range of virtual addresses. At this point, no physical memory has been allocated. Later, when the application actually touches the virtual addresses, a page fault occurs, and the operating system allocates physical memory. A page of virtual memory that has physical memory associated with it is called a ***committed*** page, whereas if no physical memory is allocated, the page is ***uncommitted***. An application or library can use `madvise()` to uncommit a committed page, returning the underlying physical memory back to the operating system.

DLmalloc [29] is the default allocator in Linux. DLmalloc is relatively simple and has been stable for decades. DLmalloc employs per-object boundary tags (which are due to

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.

ISMM'15, June 14, 2015, Portland, OR, USA
ACM. 978-1-4503-3589-8/15/06
http://dx.doi.org/10.1145/2754169.2754178

Knuth [25]). On a 64-bit machine, each allocated object is immediately preceded by 8 bytes which indicate the size of the object (and hence the address of the next object), as well as whether the object is allocated or free, and whether the previous object is allocated or free. When the previous object is free, the last 8 bytes of that object also indicate the size. When DLmalloc frees an object, it can immediately coalesce the object with the next object (if it is free) since it knows the size of the newly freed object. If the previous object is freed, it can also coalesce the newly freed object with the previous one. The boundary tags can contribute up to a 50% space overhead when allocating small (8-byte) objects. DLmalloc employs a first-fit heuristic and has no provision for multithreading beyond using a single lock to protect the entire allocation data structure. For example, DLmalloc has no per-thread caching.

This study used DLmalloc 2.8.6, which has been placed in the public domain.

Hoard [4] was introduced during the era of the first multithreaded allocators. The original Cilk allocator [5] and the STL allocator of the time [39] provided per-thread allocation, which was very fast (no locking was required), but also produced unbounded memory blowups. The problem can occur, for example when one thread allocates objects and a second thread frees them. The thread cache for the second thread can end up with an unbounded number of objects. Some allocators, such as PTmalloc [19] and LKmalloc [28] exhibit memory blowup bounded by P, the number of processors. Hoard provides a provable bound on its space blowup by moving objects out of its per-thread caches under certain conditions. Hoard organizes small objects into ***chunks***, which contain objects all the same size.[1] Large objects are dealt with using heavier-weight mechanisms. Hoard puts metadata at the beginning of each superblock indicating the size of the objects in the superblock. Since the objects within a superblock are all the same, Hoard does not need boundary tags.

Hoard uses an ***allocate-fullest*** heuristic. Hoard sometimes finds itself in a situation where an object could be allocated out of any one of several superblocks. Hoard allocates the object out of the fullest such superblock. By allocating into the fullest superblock, Hoard improves the chances that a superblock will completely empty and become available to be uncommitted or reused for other purposes.

This study used Hoard 3.10, which is licensed under GPLv2.

JEmalloc [14] is, in my experience, one of the best allocators now available for production. JEmalloc ships with BSD and Mozilla, as well as with MariaDB in recent Red Hat distributions. JEmalloc strives to minimize footprint in long-running processes such as browsers and database servers. Like Hoard, JEmalloc allocates large chunks of memory containing objects all the same size.

Allocator	Bytes	LoC
DLmalloc [29]	221K	6,280
Hoard [4]	429K	17,056
JEmalloc [14]	624K	22,230
TBBmalloc [27]	350K	9,705
SuperMalloc	127K	3,934

Figure 1. Code sizes measured in bytes and in Lines of Code (LoC). Note: TBBmalloc is tough to measure since it's mixed into TBB. SuperMalloc could end up with more code as it becomes production-quality.

Whereas Hoard uses an allocate-fullest heuristic, JEmalloc uses a lowest-address heuristic: Of all the possible objects to allocate, JEmalloc allocates the one with the lowest address. This strategy is something akin to a first-fit strategy, except that all the objects are the same size. Although fullest fit seems, in principle, to allow more blocks to be uncommitted, first fit for fixed-size objects is generally pretty good at freeing up entire blocks, and in practice seems to be just as good as fullest fit. In an earlier version of JEmalloc, Evans used fullest fit on groups of equal-sized objects, but later abandoned it. When JEmalloc returns memory to the operating system it unmaps an entire chunk — there are several groups per chunk — and fullest fit on groups does not correspond to fullest fit on chunks. Evans found that, although individual groups of equal-sized regions might benefit from fullest fit, chunks as a whole did not drain as well because a single "fullest" page could keep a chunk mapped indefinitely [16].

JEmalloc uses a thread-local cache, and eventually returns objects from the thread-local cache into the globally accessible data structure. JEmalloc seems slightly faster than Hoard in practice, and although JEmalloc does not provide any explicit bounds on memory blowup, it seems to be the allocator of choice for long-running multithreaded processes that need to keep their footprint under control.

JEmalloc uses three size categories: small, large, and huge, and implements them differently. For small and large objects, JEmalloc carves a chunk into page runs using a red-black tree to track arbitrary contiguous page runs[2] and maintains metadata at the beginning of the chunk. By storing metadata at the beginning of the chunk, the pages themselves can remain untouched and uncommitted.

JEmalloc directly maps huge objects using `mmap()`. Huge objects are chunk-aligned. JEmalloc maintains another red-black tree mapping huge chunk addresses to metadata.

[1] Chunks are called "superblocks" in [4]. They are called "chunks" in [14] and [27].

[2] Previously JEmalloc employed a buddy algorithm to track runs within a chunk, but changed to red-black trees in 2006–2007. More recent versions, which I haven't benchmarked, switched to employing a lock-free radix tree rather than a red-black tree to track huge allocations [16].

This study used JEmalloc 3.6.0, which is licensed under a variant of the modified BSD license.

TBBmalloc [27] is the allocator shipped with Intel's Threading Building Blocks (TBB) and is based on ideas developed in McRT-malloc [21]. TBBmalloc is about as fast as SuperMalloc, but has an unbounded space blowup in theory and in practice. TBB uses a thread-private heap, and never returns space for small objects to the operating system. TBBmalloc always allocates out of a per-thread pool with no locking. If the same thread returns the object as allocated it, then no locking is needed. Otherwise the object is returned to a ***foreign*** block and requires lightweight synchronization to place the object into a linked list that will be dealt with the next time the owner of the foreign block tries to allocate. Like JEmalloc and Hoard, TBBmalloc uses chunks each containing objects of homogeneous size, and places metadata at the beginning of each chunk.

TBBmalloc can have an unbounded footprint. One case was documented by [40]. In this case, one thread allocates a large number of objects, and a second thread then frees them, placing them into the first threads foreign block. If the first thread then does not call `free()`, then the memory will never be removed from the foreign block to be reused. There appears to be no easy fix to this problem in TBBmalloc, since the thread-local locking policy assumes, deep in its design, that every thread calls `free()` periodically.

This study used TBBmalloc 4.3, which is licensed under GPLv2.

Other allocators: Many other allocators exist.

For example, [34] describes a lock-free allocator based on Hoard. Streamflow [38] is also lock-free. I cannot handle the complexity of lock-free programming for anything complicated: I can write reliable lock-free code to push an object onto a linked list, but popping a list is trickier.

The VAM allocator [17] attempts to improve cache locality, especially on long-running jobs. Although VAM and Hoard share an author, the VAM paper does not compare VAM to Hoard, nor does the VAM software appear to be on the web, so I could not compare SuperMalloc to VAM.

2. SuperMalloc

Section 1 reviewed the competition, and this section explains how SuperMalloc works.

Like the other chunk-oriented allocators (Hoard, JEmalloc, and TBBmalloc), SuperMalloc allocates large chunks of homogeneous-sized objects for small objects and uses operating-system-provided memory mapping for larger objects. SuperMalloc's chunks are multiples of 2 MiB and are 2 MiB-aligned, which corresponds to huge pages on x86. In operating systems that support huge-pages, chunks align properly with the paging system. For small objects, SuperMalloc puts several objects per chunk. For larger objects, the allocation is backed by one or more contiguous chunks.

Unlike the other chunk-oriented allocators, SuperMalloc does not divide its chunks into smaller pieces. For example, JEmalloc divides its chunks into runs, and each run can contain objects of a different size. In SuperMalloc, the entire chunk is the same size. The rationale for this decision is that on a 64-bit machine, virtual-address space is relatively cheap, while physical memory remains dear. The other allocators' approach of dividing up a chunk saves virtual-address space by requiring fewer chunks to be allocated. In the case where that strategy actually does save space, it is because the application did not actually allocate a whole chunk's worth of objects of a given size. In that case, in SuperMalloc, the rest of the block is an untouched uncommitted "wilderness area" (a term apparently coined by [26]) using no physical memory.

Like JEmalloc, SuperMalloc employs several object sizes. The sizes are organized into bins. In SuperMalloc, there are small, medium, large, and huge objects. The first 45 bins are specific sizes, and larger bin numbers encode the number of pages that the application actually allocated (so that when SuperMalloc maps a 2 MiB chunk to support a 1.5 MiB request, it can properly track the amount of physical memory that the application has asked for). A 4-byte bin number allows us to allocate objects of up to just under 2^{32} pages (2^{44} bytes).

Given a pointer to an object, a chunk-based allocator must be able to determine which chunk an object belongs to, and what the size of the objects in that chunk are. For example, in JEmalloc, depending on the size of the object, the object might be looked up in a global red-black tree (for huge objects), or in a local red-black tree at the beginning of a chunk.

SuperMalloc adopted a simpler strategy. An entire chunk is all the same size objects, and SuperMalloc can look up the chunk number in a table. Instead of using a tree, SuperMalloc uses an array to implement that table, however. Since the usable x86 address space is 2^{48} bytes and chunks are 2^{21} bytes, there are only 2^{27} entries in the table. Each entry is a 4-byte bin number. The SuperMalloc table consumes 512 MiB of virtual address space, but since the operating system employs a lazy commit strategy, it typically needs only a few of pages physical memory for the chunk table. This is another instance of the design principle for 64-bit software that "it is OK to waste some virtual address space."

3. Small Objects

Small objects are as small as 8 bytes, and increase in size by at most 25% to limit internal fragmentation. Small object sizes are regularly spaced: the sizes are 8, 10, 12, 14, 16, 20, 24, ..., 224, and their sizes take the form $k \cdot 2^i$ for $4 \leq k \leq 7$ and $1 \leq i \leq 5$. In other words, a small object size, when written in binary is a 1, followed by two arbitrary digits, followed by zeros. Thus bin 0 contains 8-byte objects, bin 1 contains 10-byte objects, and so forth. To compute the bin

```
int size_2_bin(size_t s) {
  if (s <= 8) return 0;
  if (s <= 320) {
    // Number of leading zeros in s.
    int z = clz(s);
    // Round up to the relevant
    // power of 2.
    size_t r = s + (1ul<<(61-z)) -1;
    int y = clz(r);
    // y indicates which power of two.
    // r shifted and masked indicates
    //  what the low-order bits are.
    return 4*(60-y)+ ((r>>(61-y))&3);
  }
  if (s <= 448) return 22;
  if (s <= 512) return 23;
  ...
  if (size <= 1044480) return 45;
  return 45 + ceil(size-1044480, 4096);
}
```

Figure 2. Code to convert a size to a bin number. The first 22 bins are handled by bit hacking. The `clz()` function returns the number of leading 0 bits in its argument. Bins 22–45 are handled by a case statement (the elided sizes are 704, 832, 1024, 1088, 1472, 1984, 2048, 2752, 3904, 4096, 5312, 7232, 8192, 10048, 14272, 16384, 32768, 65536, 131072, 258048, and 520192.) Huge bins are computed as a constant plus the number of pages used by the object.

number from a small size can be done with bit hacking in $O(1)$ operations using the code shown in Figure 2.

For small objects, SuperMalloc makes no particular effort to avoid false sharing [6]. Like JEmalloc, SuperMalloc assumes that users who care about false sharing will explicitly ask for cache-line alignment, using e.g., `memalign()`. This decision stands in contrast to allocators, such as Hoard, that try to use temporal locality to induce spatial locality: Hoard has heuristics that try to place objects that were allocated at the same time by the same thread onto the same cache line, and objects that were allocated by different threads on different cache lines. As explained below, SuperMalloc takes a different tack on avoiding false sharing, focusing on objects that are larger than a cache line.

SuperMalloc employs a fullest-fit algorithm; when allocating an object, it finds the fullest non-full page that holds objects of that size.[3] In theory this should help empty out nearly empty pages so that they can be returned to the operating system. As mentioned above, JEMalloc abandoned fullest fit on subchunks because it was not conducive to unmapping an entire chunk. The decision to try to unmap an entire chunk makes sense on a 32-bit machine, where virtual memory space is about as expensive as physical memory, but on a 64-bit machine, uncommitting the group without unmapping the chunk would be good enough. In contrast, SuperMalloc was simplified by the decision to support only 64-bit machines — SuperMalloc only uncommits pages and chunks, and never unmaps or changes the size of the objects in a chunk.

To implement fullest fit, SuperMalloc provides, for each object size, a heap-like data structure to sort pages by how full they are. This data structure solves a simpler problem than the general priority-queue problem because pages have a bounded number of different fullness states. For example, pages containing 64-byte objects can be in only one of 65 different states corresponding to 0 objects free, 1 object free, and so forth up to 64 objects free. For each size class, SuperMalloc simply maintains an array of length 65 containing linked lists of pages, along with a bitmap to indicate which sizes have nonempty linked lists. The linked list cells are kept outside the pages so that the pages can be uncommitted when they become empty.

SuperMalloc reserves the first few pages of each 2 MiB chunk for bookkeeping. The bookkeeping comprises a bitmap of the free objects for each page and the heap-like data structure for implementing the fullest-fit heuristic (including the list cell for each page). A metaprogram generates the data structures so that, for example, the interal arrays are properly sized for each object-size class.

SuperMalloc's small objects are not all suitably aligned, and in an upcoming release I expect to remove the objects of size 10, 12, 14, 20, and 28 (leaving small sizes 8, 16, 24, 32, and then continuing with the current scheme that limits internal fragmentation to 25%). The disadvantage of doing so is that it increases internal fragmentation. For example, currently, when allocating a string of size 9, SuperMalloc returns an object of size 10, but after making this change SuperMalloc will return an object of size 16. Here I will discuss the rationale for this decision, starting with a discussion of what the standards require and of the current practice.

The C11 standard [23] is widely interpreted to mean that `malloc()` must return objects aligned suitably for any data type. On modern x86-64 processors, this appears to mean that all objects must be 32-byte aligned, since the `_m256` data type is 32-byte aligned [33]. Arguably, small objects could use less alignment. For example, any object less than 16 bytes long would need to be only 8-byte aligned, since all the objects that require 16-byte alignment are at least 16 bytes long. DLmalloc, the most commonly used allocator in Linux, returns only 8-byte-aligned objects, necessitating the use of `memalign()` when using SSE and AVX data types. A long discussion of this issue and how Linux works around the 8-byte-aligned practice, can be found at [37]. Some allocators, such as JEmalloc, return 16-byte aligned

[3] As we'll discuss in Section 4, these 'pages' are in some cases more than 4 KiB, but for now it suffices to imagine that they are ordinary 4 KiB pages.

```
struct line {
  int length;
  char contents[0];
};

struct line *thisline =
  malloc(sizeof(struct line) +
  this_length;
thisline->length = this_length;
```

Figure 3. A frequently-seen non-standards-compliant idiom is to allocate a structure with a string at the end, and then call `malloc()` with the size of the structure plus the size of the string.

objects for objects of size 16 or larger. As far as I know, no Linux allocator returns 32-byte aligned objects.

On careful reading, the standards [23, 33] do not actually appear to require 8-byte alignment for a 9-byte allocation, however. It turns out that "alignment" is not actually defined in the standard. No compliant C program can determine the alignment of a pointer. The x86-64 ABI specification [33] states that all structures are aligned according to the highest alignment required by any of their elements, and all structures' sizes are a multiple of their alignment. So according to these standards, it is fishy to cast an 11-byte alignment to a pointer to a 2-byte-aligned structure; such usage is probably not compliant. If all codes followed these standards, it would be OK to return a completely unaligned pointer when an application called for a 9-byte alignment.

Real codes do not follow the rules, however. For example, one common idiom is to write something like the code shown in Figure 3. This example is taken from the GCC manual [18]. This code uses a zero-length array as the last element of a structure, where the structure is really a header for a variable-length object. Since the `length` field is a 4-byte-aligned integer, the whole structure must be 4-byte aligned, and we might see a call to `malloc(11)` specifying 11 bytes (4 for the integer, and 7 more for the array). Because of this idiom, `malloc()` must return objects that are aligned to fit the biggest object that could fit inside.

Thus there are three possible options for a mallocator.

- Follow the standards strictly, which appear to allow 10-byte objects to be packed tightly. This will break idiomatic codes however.
- Follow the standards strictly, which appears to require any 8-byte-and-larger objects to be 8-byte aligned, 16-byte-and-larger objects to be 16-byte aligned, and 32-byte-and-larger objects to be 32-byte aligned. This approach wastes memory space, but allows all x86-64 ABI-compliant codes to run without using `memalign()`.
- Follow the Linux practice of providing 8-byte alignment.

I plan to switch SuperMalloc from the first option to the last option.

4. Medium Objects

Medium objects are all sized be a multiple of 64 (the cache line size). The smallest is 256 bytes, and the largest is 14272 bytes. The sizes chosen for SuperMalloc include powers of two and prime multiples of cache lines. The powers of two (256, 512, 1024, 2048, 4096, 8192) are used only when the application explicitly asks for aligned data. The prime multiples of cache lines are used for all other requests to reduce the contention in the low-associativity processor caches. For example, recent Intel processors provide 8-way associative level 1 cache. Following the lead of [1], who propose ***punctuated arrays*** to reduce associativity conflicts, I wanted to avoid aligning objects into the same cache associativity set. (Apparently punctuated arrays are appearing in most of the Solaris 12 allocators [11].) SuperMalloc, however, took the more "radical" approach mentioned by [10] of making all the size classes a prime multiple of the cache line, which not only reduces conflicts within a class size but reduces inter-size cache-index conflicts.

This approach introduces one big gap between 7 and 11 cache lines introducing internal fragmentation of up to 57% for an allocation of 449 bytes, and slightly smaller gap between 13 cache lines and 17 cache lines (31% fragmentation for 833-byte allocations). To close these gap, SuperMalloc added two more bins at 9 cache lines and 15 cache lines, with the rationale that a little bit of inter-size associativity conflict is better than internal fragmentation.

One aspect that becomes relatively more important for medium objects than small objects is what happens when the page size is not a multiple of the object size. Consider for example the 832-byte size bin (832 bytes corresponds to 13 cache lines). If we fill up a 4096-byte page with these objects, we would be able to fit 4 objects into a page wasting 768 bytes at the end.

Supermalloc avoids this internal fragmentation by using larger logical pages, called ***folios***, for allocation bookkeeping. For example, in the 832-byte bin, rather than wasting those 768 bytes, SuperMalloc allocates objects out of a folio containing 13 pages (53,248 bytes). When SuperMalloc allocates an 832-byte object, it finds the fullest 13-page folio (among all folios holding 832-byte objects) that has a free slot, and allocates that slot. When a folio becomes empty, SuperMalloc uncommits the entire folio, rather than trying to uncommit individual pages of a folio. There is still fragmentation at the end of the 2 MiB chunk (5 unused pages in this case), but that fragmentation comprises whole pages which are never touched and remain uncommited. Once again, I do not mind wasting a little virtual address space.

Except for the way their sizes are chosen, medium and small objects are managed exactly the same way (with folios and the fullest-fit heuristic, for example) by exactly the same code.

5. Large Objects

Large objects start at 16 KiB (4 pages) and go up to half a chunk size. Large objects are all a multiple of a page size, and the code is a little bit simpler than for small and medium objects, since it mostly does not matter which large object is returned by `malloc()`; when a large object is freed its pages can be uncommitted, so SuperMalloc does not need to keep track of anything for a fullest-fit heuristic.

In order to avoid associativity issues, SuperMalloc adds a random number of cache lines, up to a page's worth, to the allocation, and adjusts the returned pointer by that random number. For example when asked to allocate $20,000$ bytes, SuperMalloc picks a random number from 0 to 63 and adds that many cache lines to the request. If, for example, SuperMalloc picked 31 for the random number, it would allocate $21,984 = 20,000 + 31 \cdot 64$ bytes. That would be allocated out of bin 40, which contains 32 KiB objects. Instead of returning a pointer to the beginning of the 32 KiB object, SuperMalloc returns the pointer plus 1984 bytes. This strategy wastes at most an extra page per object, and since this strategy is employed only on objects that are 4 pages or larger, it can result in at most 25% internal fragmentation (which matches the fragmentation we were willing to tolerate for the small objects), and on average wastes only 12.5%. If the application explicitly asks for a page-aligned allocation, SuperMalloc skips the random rigamarole. Adding a random-offset to what would otherwise be a page-aligned allocation appeared in [31] for a page-per-object allocator, and is applied in a more systematic fashion by [1].

6. Huge Objects

Huge objects start at half a chunk (about 1 MiB) and get larger. The SuperMalloc huge object allocation primitive is straightforward, and is similar to many other allocators. SuperMalloc use operating system primitives such as `mmap()` to create new huge chunks, and never gives the memory back using `unmap()`. SuperMalloc again takes advantage of the virtual-addresses-are-cheap observation, as follows.

SuperMalloc rounds up huge objects to the nearest power of two. For example to allocate a 5 MiB object, it allocates an 8 MiB region (which is 2 MiB-aligned) of virtual memory using `mmap()`.

SuperMalloc keeps a linked list for each such power of two, so that when we `free()` a huge chunk, it add the chunk to the linked list. SuperMalloc actually threads the linked list through the chunk table so that the chunk itself does not need to be in committed memory. To free a huge object SuperMalloc explicitly uncommits the memory using `madvise()`.

SuperMalloc uses `madvise()` to ask the kernel to map the huge chunk using huge pages (recent Linux kernels support 2 MiB pages, which can reduce TLB pressure and reduces the cost of walking the page table in the case of a TLB miss [7]). If the object is not a multiple of 2 MiB in size, the last fractional part of the huge page is mapped only with normal 4 KiB page table entries. Some users do not like transparent huge pages, and if a user wants fine-grained commitment within a huge object, they must call `madvise(MADV_NOHUGEPAGE)` to ensure the small pages.

Thus, although SuperMalloc allocated 8 MiB to implement a 5 MiB allocation, only 5 MiB would actually ever be committed to memory (and then only when the application code touches it.

For small, medium, and large objects, SuperMalloc makes no explicit use of huge pages, meaning that on most systems huge pages are not used for those non-huge objects. On most Linux systems, huge pages are enabled only when explicitly asked for with madvise, e.g., with

```
$ echo madvise > /sys/kernel/mm/\
  transparent_hugepage/enabled
```

and on such systems the non-huge chunks will all be mapped with 4 KiB pages. I hope to make better use of huge pages in the future for SuperMalloc. One idea is to do the bookkeeping to keep track of when a chunk becomes, say, 90% allocated, and then enable huge pages on that chunk. Part of the problem is that as soon as we uncommit a single 4 KiB page using `madvise(MADV_DONTNEED)`, then the entire chunk becomes ineligible for huge pages. The allocator would need to defer the uncommit operations, and then when, say, only 80% of the pages are in use, run all the deferred uncommit operations. The system would need some hysteresis so that it would not bounce between huge-page and no-huge-page mode too frequently.

7. Arithmetic

One common operation in SuperMalloc is to calculate indexes in a bitmap. For example, given a pointer p, the code must perform calculation in Figure 4(a) to calculate to which folio and which object in the folio p refers.

Since the chunk size is a power of two, computing the chunk number is easy. Computing the folio number requires dividing by the folio size, which isn't a power of two, and isn't known at compile time, however. Division is slow, so SuperMalloc follows the approach of [32] to convert division to multiplication and shift. For example, for 32-bit values of `x`, we have

```
x / 320 == (x*6871947674lu)>>41.
```

Figure 4(b) shows the faster code. SuperMalloc uses metaprogramming to generate all the object sizes and magic numbers, for example, to find prime numbers that are spaced no more than 25% apart, and to calculate the multiplication and shift constants to implement division..

```
1  C = p / chunksize;          // Compute chunk number.
2  B = chunkinfos[C];          // What is the bin?
3  FS = foliosizes[B];         // What is the folio size?
4  FN = (p%chunksize)/FS;      // Which folio in the chunk?
5  OS = objectsizes[B];        // How big are the objects?
6  ON = ((p%chunksize)%FS)/OS; // Which object number in the folio?
```

(a)

```
1  C = p / chunksize;
2  B = chunkinfos[C];
3  FS = foliosizes[B];
4  FN = ((p%chunksize)*magic0[B]) >> magic1[B];       // magic
5  OS = objectsizes[B];
6  ON = ((p%chunksize-FN*FS)*magic2[B]) >> magic3[B]; // magic
```

(b)

Figure 4. The calculation to compute the folio number in the chunk, `FN`, and the object number in the folio `ON`, so that the bitmap for the free objects in the folio can be updated. (a) shows the code with expensive divisions in lines 4 and 6. (b) shows the code with the divisions replaced by multiplication and shift.

8. Caching

Like other multithreaded allocators, SuperMalloc employs per-thread caching. In addition to the per-thread cache, SuperMalloc also employs a per-CPU cache, as well as a global cache in front of the true data structures. Each cache is size-segregated (that is, there is a cache for each size bin), but we will describe it here as though there were only one size. Each cache comprises a small number of doubly-linked lists of objects that belong in its bin. A single doubly-linked list can be moved from one level of the cache to the other in $O(1)$ operations, which reduces the length of the critical section that moves the list.

The cost of locking and accessing a shared data structure turns out to be mostly due to cache coherence traffic from accessing the object, rather than the locking instructions. Figure 5 shows the overheads for incrementing a global variable protected by a global lock when multiple threads are running. For example on our E5-2665, that was 485.5 ns. One alternative is to use a thread-local variable which costs only 3.1 ns. Most modern allocators use a thread-local cache of global values. But it turns out that a per-CPU data structure costs only 34.1 ns to access. The difference is that the per-CPU data structure mostly resides in the right cache (it's in the wrong cache only when the operating system migrates a thread to a different processor), and the lock instruction is mostly uncontended (contention happens only when the operating system preempts a thread). So instead of paying for all the cache-coherence traffic, one pays only the cost of executing the lock instruction and the increment instruction. For the per-CPU cache the system also must determine which CPU is running which SuperMalloc determines using a call to `sched_getcpu()`. If we factor out the cost of `sched_getcpu()`, the per-CPU data structure costs only 13.3 ns. Even with the call to `sched_getcpu()`, the per-CPU structure is fast. Based on these numbers, I concluded that a per-CPU cache gets most of the advantage of a per-thread cache, but that SuperMalloc still needs a small per-thread cache. Uncontended locking is not so bad.

A related approach used by some systems is to hash or otherwise map the thread identifier to a cache number instead of using a per-CPU cache (e.g., [14, 28]). The cache number is then used to index a collection of not-quite-thread-and-not-quite-per-CPU caches. This hashing-or-mapping approach typically requires more caches than there are CPU's, however, and can be analyzed as follows. If there are P threads running,[4] P processors, and P caches, we still expect quite a bit of contention on some of the caches. In fact, we expect only about P/e threads to operate without contention due to a standard balls-and-bins argument (and some cache will have contention about as bad as $\log P/\log\log P$). Since the cost of contention is so high, this would reduce the cost of access to no less than

$$485.5\text{ ns}/e + 34.1\text{ ns}\cdot(1-1/e) = 212\text{ ns},$$

which is still an order of magnitude slower than the uncontended per-CPU operation. To get the contention close to zero, we can apply the birthday paradox and conclude that we might need P^2 caches. That's probably overkill, since if we got the contention down to, say $1/20$ of the accesses, we would get rid of enough contention to get to within about half of optimal. To get the contention to $1/20$ requires approximately $3P$ caches. (JEmalloc uses $4P$ caches.) One advantage of using only P per-CPU caches instead of $3P$ (or P^2) caches indexed by a hash of the thread index, is that

[4] There may be more threads actually running, but at least for a scheduling quantum, only P of them are actually scheduled.

	i7-4600U 2 cores 2.1 GHz	i7-4770 4 cores 3.4 GHz	E5-2665 16 cores 2 sockets 2.4 GHz
Global	149.0 ns	120.6 ns	485.5 ns
Per-CPU	29.6 ns	18.0 ns	34.1 ns
Per-CPU no getcpu	17.2 ns	10.6 ns	13.3 ns
Per-thread	3.3 ns	2.1 ns	3.1 ns

Figure 5. Lock vs. contention overhead. "Global" is the cost of accessing a global lock and incrementing a global counter. "Per-CPU" is the cost of call `sched_getcpu()` and accessing a per-cpu lock and increment a counter. "Per-CPU no getcpu" is same thing without the call to `sched_getcpu()`. Per-thread is the cost of incrementing a thread-local variable without a lock .

the cache takes less space. All the objects in the caches are likely to be using committed memory, and we want to waste as little physical memory as possible. Another advantage is that the software cache is likely to be in the right hardware cache to allow the processor to access it without cache misses. The biggest disadvantage of per-CPU caches is calling `sched_getcpu()`, which is not very expensive, and can be reduced by calling it less frequently (say, only 1/16th of the time): if `malloc()` is being called frequently, it will be using the right CPU most of the time, and if called infrequently, the performance will not matter much.

The SuperMalloc per-thread cache needs to be just big enough to amortize the 34 ns uncontended locking overhead compared to the 3.1 ns unlocked overhead, so we want about 10 objects in the per-thread cache. I decided to store two linked lists each containing up to 8 KiB worth of objects for each bin in the thread cache.

I tried to tune the per-CPU cache to be just big enough to fill up the processor's data cache. I reasoned that that if the working set of a thread is bigger than the data cache, the thread will be incurring cache misses anyway. In this case, there is no point in working hard to avoid a few more cache misses. In contrast, many allocators keep many megabytes in their per-thread caches. Because there are many per-thread caches (thousands of threads in a typical database server, for example), their footprint is larger. I chose to store two linked lists each containing up to 1 MiB for each bin in the per-CPU cache.

There is also a global cache containing P linked lists of up to 1 MiB each.

The logic for allocating an object is as follows:

1. If the thread cache has an item, use it. This requires no locking.
2. Otherwise if the CPU cache contains a non-empty linked list, move it to the thread cache, and proceed to step 1. Moving an item from the CPU cache requires mutual exclusion, which is implemented with a transaction on chips that support transactions (such as Haswell) otherwise with locking. There is a lock for each cpu-bin pair.
3. Otherwise if the global cache contains a non-empty linked list, move it to the thread cache, and proceed to step 1. This step also requires mutual exclusion (with a transaction or a lock), and there is a lock for each bin.
4. Otherwise access the global data structures, which requires mutual exclusion (with a transaction or a lock), and there is a lock for each bin.

The logic for deallocating an object is similar:

1. If one of the linked lists in the thread cache isn't "full", then add the object to that list.
2. Otherwise, if one of the CPU caches is empty, move one of the linked lists from the thread cache to the CPU cache (add the object to that list while we are at it.)
3. Otherwise, if one of the global caches is empty, move one of the linked lists from the thread cache to the global cache (add the object to that list while we are at.)
4. Otherwise access the global data structures to free the object.

Dynamically, most of the critical-sections are on the CPU cache, and since there is no guarantee that we stay on the same CPU for the duration of a critical section, we need locks. The CPU cache employs a lock for each CPU-bin pairing. Since there are 45 bins and 32 CPU's on the biggest machine I measured, that's 1,440 locks in use. I placed each lock on its own cache line. I considered placing the locks and the doubly-linked list heads into the same cache line, but concluded that that might cause performance trouble. For example, on the HTM hardware, the code spins on the lock before starting the transaction. But that spin could cause a transaction to fail if the lock were on the same cache line as the data being modified. Without HTM, spinning on the lock could interfere with the updates being made to the linked lists by causing the cache line to bounce around.

For the global data structure, SuperMalloc has a lock for each bin. Since the code accesses the global data structure relatively infrequently, the lock is mostly uncontended.

When running on hardware that supports HTM, SuperMalloc uses transactions. If the transaction fails enough, it falls back to locking code using the original lock structures. To make the HTM transactions interact properly with the lock-protected transactions, the HTM transaction must look at the lock state, and verify that it is unlocked. This lock examination is called ***subscribing*** to the lock. One decision to make is whether to subscribe to the lock early or late in the transaction. Subscribing late sounds enticing because it reduces the time during which the lock is contended (when in fact we might not conflict on the data structure). Subscribing late can be dangerous, however, since it can be difficult to prove that your code does not get tricked into committing [12]. SuperMalloc subscribes to the lock early in the transaction for three reasons. (1) I do not feel confident that I can guarantee that the code avoids all the pitfalls of late subscription. (2) SuperMalloc's data structures are closely related to its locks. It's unlikely that the lock conflicts but that the data does not conflict. (3) SuperMalloc's transactions are all so short that the performance impact is small; the performance advantage of late lock subscription appears to be about half a percent for high-threadcount runs of the malloc-test benchmark. In the future, I hope to show that late lock subscription is correct for SuperMalloc, since I do not want to leave half a percent on the table.

In order to make the critical sections short, both for HTM and locking, SuperMalloc tries to prefetch everything that will be accessed, then waits for the lock to become free (and if the lock wasn't free, it prefetches everything again, and checks the lock again.) The goal is to make it so that while the lock is held (or the transaction is running), as few cache misses as possible are incurred. This prefetching improves performance by a few percent, as shown below.

9. Performance

We are interested in three aspects of performance: speed, footprint, and complexity.

Speed: A poor allocator can be slow even on serial applications, and the situation can get far worse on multithreaded codes where the allocator can become a serial bottleneck. On a multicore with a multithreaded workload the speed difference between the default allocator in Linux [29] and a state-of-the-art allocator such as JEmalloc [14] can be more than a factor of 30. If the cost of allocation varies significantly it can be difficult to meet the soft real-time constraints of a server. I have seen JEmalloc causing a 3 s pause about once per day on a database server, which would be too slow for a social-networking site, for example. (I traced the problem in JEmalloc to the code inside `free()` which occasionally gathers together a large number of unused pages and uncommits them using `madvise()`. The delay is not inside the `madvise()` call, rather it is in the code that gathers together the unused pages. But I do not understand why it takes 3 s.) It was this 3 s pause that got me interested in building SuperMalloc. (I do not yet have any evidence that SuperMalloc actually solves the 3 s-pause problem.)

One of the best benchmarks for a multicore allocator is malloc-test [30]. (Malloc-test results are also reported in [14], and the cross-test of [27] is essentially the same test.) The malloc-test benchmark runs K producer threads and K consumer threads. Each producer thread allocates objects as fast as possible, and each consumer frees them. Malloc-test offers one of the most difficult workloads for multithreaded allocators that employ per-thread caching, since the per-thread caches can quickly become unbalanced. Malloc-test is a tough benchmark because per-thread caches often do not work well. The producer's caches are always empty, and the consumer's caches are always overflowing. I obtained malloc-test from XMALLOC repository [13]. It turns out that the malloc-test benchmark is showing its age: it is racy and exhibits a huge amount of variance even if the allocator being tested runs infinitely quickly. Malloc-test operates by creating an array of 4096 pointers which is shared between a producer and a consumer. The producer repeatedly looks for an NULL value in the array, an if it finds one, allocates an object and stores it. The consumer repeatedly looks for a non-NULL value in the array, and if it finds one, frees it, and sets the slot to NULL. It turns out that for fast allocators, the producers and consumers both can spend a lot of time spinning around looking for interesting slots. I improved malloc-test by having each producer create a batch of 4096 objects in a structure, which is placed into a queue protected by a pthread condition variable. If the queue gets full, the producers wait. Each consumer waits for the queue to be non-empty, dequeues one of the batches, and frees all 4096 objects. These changes not only made malloc-test run more evenly, but they also made it run much faster. Malloc-test was so slow and noisy that I could not tell the difference between any of the allocators except for DLmalloc. Sometimes benchmarks need work so that they can run fast.

As mentioned above, the cost of early lock subscription inside transactions is about half a percent for a high-threadcount run of our improved implementation of malloc-test; that is almost unmeasurable, since the standard deviation of the runtime is also about half a percent. This situation illustrates the importance of reducing the variation in runtime for benchmarks. With the original version of malloc-test the variance is more than ten percent, and we would be unable to see the advantage of late lock subscription.

Figure 6 compares the performance of SuperMalloc against DLmalloc, Hoard, and JEmalloc on the malloc-test benchmark running on a machine with transactional memory. This is a 4-core single-socket Haswell server with 8 hardware threads. The x axis is the number of producer threads, so when there are 8 producer threads, there are also 8 consumer threads and the 8 hardware threads are oversubscribed by a factor of two. DLmalloc is so slow compared to the others that it is hard to distinguish from zero—it actually slows down as the number of threads increases. Hoard holds up pretty well, especially considering its age, but starts falling behind JEmalloc when there is one pthread per hardware thread. TBBmalloc starts out scaling better than Hoard or JEmalloc, but ends up at about the same place as JEmalloc. SuperMalloc starts out faster than the others on one producer thread—simpler data structures run faster—and scales nearly linearly up to four producer threads, which is eight total threads, and the hardware runs out of parallelism.

This is one workload for which Intel's Hyperthreading technology essentially gives perfect speedup, because the code is not bound by the arithmetic performance of the CPU nor by the memory subsystem. Instead the performance appears to be limited by the rate at which the processor

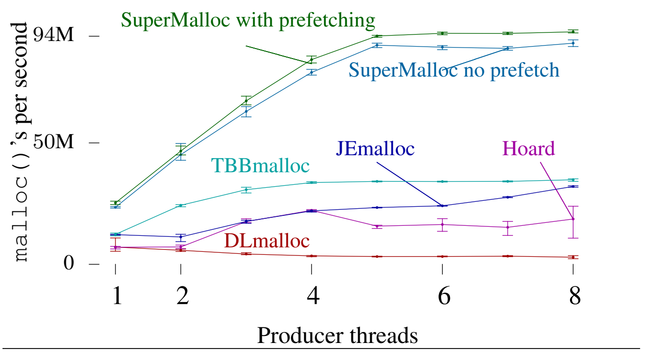

Figure 6. A comparison of SuperMalloc, DLmalloc, Hoard, JEmalloc, and TBBmalloc running malloc-test on a 4-core (1 socket + hyperthreading) 3.4GHz i7-4770 (Haswell-DT) The lines are the average of 8 trials, and the error bars show the fastest and slow trial.

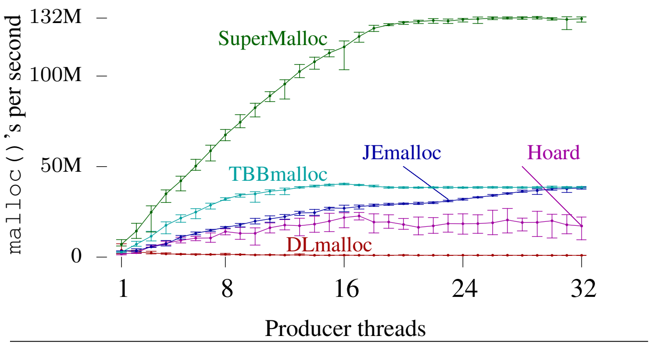

Figure 7. A comparison of SuperMalloc, DLmalloc, Hoard, and JEmalloc running malloc-test on a 16-core (2 sockets + hyperthreading) 2.4GHz E5-2665 (Sandy Bridge). The lines are the average of 8 trials, and the error bars show the fastest and slow trial.

issues memory operations that miss cache. Hyperthreading doubles that rate.

I measured SuperMalloc with the per-transaction prefetching disabled, and can see that prefetching the data needed by a transaction before the transaction starts is worth a few percent.

It turns out that the transactions fail, executing the lock-based fallback code about 0.03% of the time.

I originally designed the SuperMalloc for machines with HTM hardware. I wanted to make it highly likely that the transactions would commit. How well does the code run using pthread mutexes? It turns out that Glibc pthread mutexes are using hardware transactional memory on our machine. The performance with pthread mutexes is slightly slower than running transactional memory, but the slowdown may not be significant. I tried running the code using pthread mutexes on a Sandy Bridge machine which does not have HTM. Figure 7 shows the results. This machine has 32 hardware threads, and so SuperMalloc saturates the hardware at about 16 producer threads. It turns out that the same decisions that make HTM run fast also make lock-based code run fast.

	Mean	Fastest	Slowest	Diff
DLmalloc	340.8 ns	277.5 ns	428.8 ns	151.4 ns
Hoard	53.6 ns	41.8 ns	92.9 ns	51.2 ns
JEmalloc	31.1 ns	30.9 ns	31.5 ns	0.6 ns
TBBmalloc	28.7 ns	28.4 ns	29.2 ns	0.8 ns
SuperMalloc	10.6 ns	10.4 ns	10.7 ns	0.4 ns

Figure 8. The speed of the each allocator, measured in time, for the 8-thread case of Figure 6.

I ran the experiments compiled with gcc 4.8.3 running on Linux 3.15.7 with glibc 2.18 on Fedora 20. I compared SuperMalloc to Hoard 3.10, jemalloc 3.6.0, and tbb43_20140724.

Figure 8 explores the variance of the allocators I measured on the same experiment as for Figure 6. I use the difference between the slowest run and the fastest run as a measure of the variance. Not only are the newer allocators faster than DLmalloc, but their variance is less. In fact, the mean time to allocate for the others is less than the variance of DLmalloc. For JEmalloc, TBBmalloc and SuperMalloc, the worst-case time to allocate is less than the variance of DLmalloc. For all three of JEmalloc, TBBmalloc, and SuperMalloc, the variance is similar, but since SuperMalloc runs about three times faster, the relative variance is correspondingly three times less.

The second important benchmark measured by [14] is Super Smack, which appears to be suffering from code rot. I cloned `https://github.com/winebarrel/super-smack` into `https://github.com/kuszmaul/super-smack` and got it working. The performance is shown in Figure 9. The performance was not as interesting for these allocators as it was when Evans measured it in 2006 (in which phkmalloc looked quite bad), since all five allocators are essentially the same speed with a few percent variance. The memory footprint is interesting: DLmalloc appears to have use a few percent less memory, and TBBmalloc uses a few percent more memory. (TBBmalloc appears to suffer a memory leak; every time I run the benchmark, the footprint grows by about a megabyte.) The other allocators were about the same, each showing a few megabytes variation in the footprint.

I tried running a few of the other benchmarks mentioned by [14]. Some of the benchmarks are difficult to reproduce, and like Evans, I concluded that many of these benchmarks are not very informative. Here I discuss the ghostscript benchmark, as an example.

gs: The gs benchmark was fairly easy to replicate, at least in spirit. There is a long history of running Ghostscript: [8] ran it on an unspecified 126 page manual. [14] ran it on an unspecified file named "ps3.ps". So I ran an unspecified file[5] and ran it through Ghostscript 9.14 as

[5] If you really want to know, I used the 3251-page, 16,819,944-byte [22].

	Max RSS	speed (queries per second)
DLmalloc	131M	104K
Hoard	133M	104K
JEmalloc	137M	104K
TBBmalloc	140M	104K
SuperMalloc	135M	103K

```
$ super-smack -d mysql \
          smacks/select-key.smack 50 20000
```

Figure 9. Performance of various allocators running super-smack. The smallest footprint of five measurements is shown, and the fastest of the five measurements is shown. The Max RSS varies by several percent, but DLmalloc appears to be consistently a few megabytes smaller than the other allocators, and TBBmalloc appears to be consistently larger. The time of the various allocators also varies, and the differences appear not to be significant.

	Time$\pm\sigma$	Max RSS
DLmalloc	195.3 ± 1.0 s	150 MiB
Hoard	212.9 ± 0.6 s	135 MiB
JEmalloc	207.9 ± 0.8 s	129 MiB
TBBmalloc	194.5 ± 1.2 s	158 MiB
SuperMalloc	197.3 ± 1.4 s	161 MiB

Figure 10. The run time and maximum resident memory of running Ghostscript on [22]. The run time is the average over eight runs with the standard deviation shown as a $\pm$. The maximum resident memory is the maximum over the four runs.

```
$ gs -dBATCH -dNODISPLAY manual-325462.pdf
```

on a 4-core i7-4770 and saw the performance and maximum resident memory (RSS) shown in Figure 10. GS apparently allocates large objects and runs its own allocator internally, mostly measuring the behavior of large-object allocation. The difference between DLmalloc and TBBmalloc is small, less than one standard deviation. SuperMalloc is slightly slower, and the difference appears to be just larger than the measurement error. JEmalloc and TBBmalloc are significantly slower; it is surprising that memory allocation can have such a large performance impact, given that Ghostscript manages its own memory. SuperMalloc is, by a small margin the most wasteful of space. Interestingly, JEmalloc and Hoard, which are the slowest, also exhibit the smallest memory footprint.

Larson: I ran the Larson benchmark [28], which is included in the Hoard distribution. The Larson benchmark simulates a server in which each thread allocates objects, keeps some of them, eventually freeing them, and passes some objects to other threads where they will be freed. I ran

	Mean Time/op	σ	Max RSS
DLmalloc	1.72 µs	50 µs	6.6 MiB
Hoard	1.57 µs	38 µs	7.0 MiB
JEmalloc	1.47 µs	40 µs	8.7 MiB
TBBmalloc	1.52 µs	42 µs	9.9 MiB
SuperMalloc	1.42 µs	39 µs	7.3 MiB

Figure 11. The performance of the Larson benchmark. The time per operation is the mean over four runs, with the standard deviation. The maximum resident memory is shown. The malloc standard deviation is the variation among calls to `malloc()`.

```
./larson 10 7 500 1000 10000 1 8
```

on a 4-core i7-4770 and saw the performance shown in Figure 11. Instead of reporting throughput (which is operations per second), I report time per operation, so that we can more easily compare averages and standard deviations. I also report standard deviation of the time to call `malloc()`, and observe that the standard deviation is much larger than the mean. This means there are some very slow allocations (the slowest allocations we measured took 81 ms, and it wasn't clear that any of the allocators has any particular advantage on this metric). I was motivated by the desire to reduce the slowest allocations, but it looks like I'll have to embark on a study of much longer-running jobs to determine whether I've made progress. SuperMalloc was fastest by a small margin. This margin is much less than a standard deviation, but it appears to be repeatable.

SuperServer: I wrote another server benchmark, inspired by Larson, in which the object sizes are distributed over a wide range of sizes. For each allocation, conceptually flips coin several times and counts how many tails in a row came up, calling that number k. So $k = 1$ is chosen with probability $1/2$, and $k = 2$ is chosen with probability $1/4$, and so forth with $k = m$ chosen with probability $1/2^m$. (To avoid giving SuperMalloc an unfair advantage for small objects, which SuperMalloc aligns less rigorously than do the other allocators, I set k to a minimum value of 4.) Having picked k, the benchmark picks a random number chosen uniformly from the range 2^k (inclusive) to 2^{k+1} (exclusive). Each object's expected lifetime is proportional to its size. The object will, at the end of its life, be freed by a randomly chosen other thread. (This benchmark can be found in the SuperMalloc distribution.)

Figure 12 shows the results of this SuperServer benchmark on the five allocators tested. The fastest allocators are Hoard and TBBmalloc, with JEmalloc running close behind, followed by SuperMalloc and DLmalloc. SuperMalloc has the smallest footprint by a significant margin, with Hoard and DLmalloc off by a factor of two. TBBmalloc and DLmalloc both have an occasionally slow allocation that takes more than a millisecond, whereas the others keep the slow-

	User	System	Elapsed	Max RSS	Slowest Allocation	Slowest Free
DLmalloc	38.43 s	4.15 s	13.01 s	1833 MiB	2.534 ms	0.007 ms
Hoard	36.24 s	4.95 s	11.68 s	1853 MiB	0.534 ms	0.045 ms
JEmalloc	37.36 s	4.15 s	11.95 s	1437 MiB	0.546 ms	0.009 ms
TBBmalloc	36.45 s	4.62 s	11.52 s	1525 MiB	1.149 ms	0.025 ms
SuperMalloc	41.81 s	0.95 s	12.81 s	941 MiB	0.532 ms	0.040 ms

Figure 12. The performance of the SuperServer benchmark running on the same 4-core Haswell as in Figure 6. The numbers present the data from the run, of five runs, with the fastest elapsed time.

est allocation down to about half a millisecond. All of the allocators are pretty fast at calling `free()`.

Footprint: The ***memory footprint*** of an application, the amount of physical memory that the application consumes while running, can also vary by more than an order of magnitude — even a factor of two can be too much on something like a database server where memory is used as a cache for disk, and an increased footprint results in either a reduced effective cache size or excessive I/O's due to paging. I developed a benchmark based on the idea of [40], which operates by Thread 0 allocating data and giving it to Thread 1 which frees the data. Thread 1 then allocates data and gives it to Thread 2, which frees the data. Thread 2 then allocates and gives to Thread 3. I configured the Vyukov benchmark to allocate 50,000 random objects per thread, each object chosen to be a random size less than 4 KiB, and 200 threads. The application thus never asked for more than 100 MiB at a time. The results are shown in Figure 13. Interestingly, DLmalloc and TBBmalloc both blew up memory substantially, TBBmalloc nearly crashing our benchmark machine in the process. DLmalloc used 2.3 GiB of RSS, and TBBmalloc used 4.1 GiB. Also interesting is that JEmalloc used a great deal of system time, apparently aggressively calling `madvise()` to uncommit memory.

Complexity: A simple memory allocator can operate using only a few hundred lines of code [24]. Since allocator performance is so important, however, most allocators have been tuned to run faster at the cost of increased complexity. The code sizes of Figure 1 show that SuperMalloc is smaller, but where does the code complexity come from in these allocators?

Figure 14 shows the largest modules in SuperMalloc. The biggest module is for managing the cache. The interface to make the standard POSIX calls is second biggest (including functions such as reallocation, and testing). The code for actually performing allocation in the three size classes (small objects, medium/large objects, and huge objects) together is just a little larger than the cache management code. The code for invoking locks or hardware transactional memory, and the metaprogram that computes the bin sizes and sets up all the constants are each also about 350 lines of code. The code for interfacing to mmap to get chunk-aligned memory is 123 lines.

	User	System	Elapsed	MaxRSS
DLmalloc	15.5 s	5.8 s	26.5 s	2375 MiB
Hoard	14.7 s	0.3 s	20.1 s	284 MiB
JEmalloc	16.2 s	10.3 s	30.6 s	228 MiB
TBBmalloc	17.3 s	5.3 s	111.3 s	4214 MiB
SuperMalloc	16.0 s	0.2 s	21.3 s	219 MiB

Figure 13. The performance of the Vyukov benchmark, showing measurements of user time, system time, elapsed time, and max RSS.

Module	Characters	LoC
mmap-interface	5,066	123
huge objects	7,312	189
medium & large objects	10,203	302
metaprogram	14,791	332
atomicity and locks	9,296	357
small objects	17,096	461
Front end (API)	17,256	536
Cache management	25,340	841

Figure 14. The sizes of the significant modules of SuperMalloc, sorted by size.

For JEmalloc, the biggest module manages their "arenas" at 2,577 LoC, which implements the meat of their code, basically implementing the functionality of our objects modules and the cache management (which for us adds up to 1,793 LoC). One significant module in JEmalloc, in terms of lines of code, manages the tuning parameters (giving the user control, for example, of the number of arenas and providing access to various counters). Although that code has many lines of code, it is not complex. It's just long because there are many tuning parameters. I do not want tuning parameters in SuperMalloc, preferring instead that the code work well in all cases. It may turn out that our determination to avoid tuning parameters will fade away when faced with the demands of production — we should chalk up this part of JEmalloc's complexity to the fact that it's production-quality code.

Hoard contains a huge number of modules, each of which is only a few hundred lines of code, which can be used to build good special-purpose and general-purpose allocators

[2]. SuperMalloc isn't trying to enable to construction of special-purpose allocators, so our code is smaller.

10. An Operating-System Wish List

There are two features that Linux does not provide which are provided by some other operating systems: a cheaper way to uncommit memory, and a hook to reduce the odds that a thread is preempted while holding a lock. A third desirable feature, ***subscribable mutexes***, does not appear in any operating system, as far as I know.

Cheap uncommit: The first problem is to uncommit memory cheaply. Linux provides a function

```
madvise(ptr, len, MADV_DONTNEED)
```

which informs the operating system that the `len` bytes of memory starting at address `ptr` are no longer needed. For anonymously mapped memory (the kind that `malloc()` uses), this removes the page from the page table, freeing the physical page. The next time the process touches the page it will cause another page fault, at which point the operating system will allocate and zero-fill a physical page. That is a fairly expensive operation — it takes 1800 ns to call `madvise(MADV_DONTNEED)` to uncommit a page, whereas if the page is already uncommited it costs only 240 ns to make the same system call.

One potential alternative is to use

```
madvise(ptr, len, MADV_FREE),
```

which is provided by FreeBSD and Solaris. The semantics of this call is to give the kernel freedom to uncommit the page the way `MADV_DONTNEED` does at any time until the process writes the page again.[6]

The kernel also has the freedom not to touch the page, so that the data could still be the same data. Since the kernel is in a position to understand memory pressure as well as the overall system load, it can decide when and whether to spend the CPU cycles to reclaim memory.

Evans notes [15]

> `MADV_FREE` is *way* nicer than `MADV_DONTNEED` in the context of malloc. JEmalloc has a really discouraging amount of complexity that is directly a result of working around the performance overhead of `MADV_DONTNEED`.

I agree with Evans' sentiments, although the SuperMalloc caches reduce the impact of the `madvise()` calls. Although SuperMalloc would not get much simpler with `MADV_FREE`, it would speed up and exhibit less performance variance.

An alternative approach would be for the kernel to deliver an event to a process indicating that it should use less memory. The allocator generally has a big list of empty pages lying around that it could quickly return to the kernel. Both CRAMM [41] and Bookmarking Garbage Collection [20] discussed a modified Linux kernel that can indicate memory pressure or that a page is being scheduled for eviction, which can improve the performance garbage-collected systems under memory pressure.

Lock-aware scheduling: The second problem shows up when running in locking mode (that is, without transactional memory). Sometimes while a lock is held by a thread, the kernel preempts the thread and schedules something else. Since the thread holds a lock, any other threads that try to acquire that lock suspend too, which can lead to lock convoying and other performance problems.

Solaris provides one way to fix this using the `schedctl()` system call, which tells the kernel not to, if possible, preempt a thread for a little while[9]. The mechanism includes two steps: (1) the thread says it is about to enter a critical section, and (2) the thread indicates that it has left the critical section. Step (1) is accomplished by setting a bit in a thread-specific structure. Meanwhile the kernel, if it wanted to preempt the thread, sets a bit in the structure. Step (2) is accomplished by checking to see if the "wanted-to-preempt" bit is set, and if so, politely yielding. The kernel must take some care to downgrade threads that abuse the mechanism.

The Linux maintainers seem skeptical of this feature. For example, see [3], which proposed the patch and claimed that it improves TPC-C performance by 3%–5%, but the kernel maintainers dismissed as "a feature for a real-time kernel" and suggested a voluntary preemption model instead, and noted that they were "skeptical about the whole idea". One proposed [35, 36] that a user-space mechanism to set the priority of a thread would solve the problem. As an implementer of `malloc()`, I do not see how it would — `malloc()` is in a library, and I do not know what, if any, thread priority schemes the application may be using.

I would like to see a hook in Linux to avoid preempting threads that are briefly holding a lock.

Subscribable mutexes: When running with HTM, SuperMalloc waits until the lock used for the fallback path is not held, then subscribes to the lock. It is not difficult to subscribe to the pthread mutex lock in Linux, if you are willing to violate the POSIX abstraction. There is a single field in the mutex object that indicates whether the lock is held. It would better if the mutex provided a way to subscribe to the lock abstractly.

A more difficult problem is what the code should do when waiting for a mutex to become free. We do not want to spin indefinitely, but there is no way to wait ask the operating system to wake us up when the mutex becomes unlocked, except to lock the mutex. What we need is a way to wait for the mutex to become free without locking the mutex.

[6] Evans notes [16] that the page-table bit that tracks whether a page is reclaimable gets cleared only during a write, whereas the BSD and Solaris manual pages refer to "the next time the page is referenced". This subtle difference can result in difficult-to-debug application errors.

11. Conclusion

SuperMalloc compares well to the other scalable allocators. Hoard and JEmalloc do pretty well under all loads, whether you measure time, variance, or maximum RSS. Under difficult high-contention workloads or workloads in which the objects vary in size, SuperMalloc appears to scale better, use less physical memory, all while using less code.

SuperMalloc is available from `https://github.com/kuszmaul/SuperMalloc`, licensed under either the Apache 2.0 or GPLv3 license.

Acknowledgments

Jason Evans helped me understand the role of alignment, explained the subtleties of `MADV_FREE`, prodded me to more carefully explain how I deal with huge pages, and corrected several misconceptions I had about how JEmalloc works. Emery Berger helped me better understand how Hoard works and directed my attention to [20, 41]. Michael A. Bender helped me understand how to implement the fullest-fit priority queue. Keith Takeda proofread paper; he finds missing article and disagreeing verb. This research was supported in part by Foxconn, and in part by NSF grants CCF-1314547, CNS-1409238, and IIS-1447786.

References

[1] Y. Afek, D. Dice, and A. Morrison. Cache index-aware memory allocation. In *Proceedings International Symposium on Memory Managment (ISMM)*, pages 55–64, San Jose, California, June 2011. doi:10.1145/2076022.1993486.

[2] A. Alexandrescu and E. Berger. Policy-based memory allocation. Dr. Dobb's, Dec. 1 2005. `http://www.drdobbs.com/184402039`. Viewed Apr. 27, 2015.

[3] K. Aziz. Pre-emption control for userspace, Mar. 3 2014. `http://lkml.iu.edu/hypermail/linux/kernel/1403.0/00780.html`. Viewed Apr. 27, 2015.

[4] E. D. Berger, K. S. McKinley, R. D. Blumofe, and P. R. Wilson. Hoard: A scalable memory allocator for multithreaded applications. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 117–128, Cambridge, MA, Nov. 2000. doi:10.1145/378993.379232.

[5] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. In *Proceedings of the IEEE Symposium on Foundations of Computer Science*, pages 356–368, November 1994. doi:10.1145/324133.324234.

[6] W. J. Bolosky and M. L. Scott. False sharing and its effect on shared memory performance. In *Proceedings of the USENIX Symposium on Experiences with Distributed and Multiprocessor Systems (SEDMS IV)*, pages 57–71, 1993.

[7] J. Corbet. Transparent huge pages in 2.6.38, Jan. 11 2011. `http://lwn.net/Articles/423584/`. Viewed Apr. 27, 2015.

[8] D. Detlefs, A. Dosser, and B. Zorn. Memory allocation costs in large c and c++ programs. *Software Practice and Experience*, 24(6):527–542, June 1994. doi:10.1002/spe.4380240602.

[9] D. Dice. Inverted schedctl usage in the JVM. David Dice's Weblog, June 11 2011. `https://blogs.oracle.com/dave/entry/inverted_schedctl_usage_in_the`. Viewed Apr. 27, 2015.

[10] D. Dice. `malloc` for Haswell — hardware transactional memory. David Dice's Weblog, Apr. 24 2014. `https://blogs.oracle.com/dave/entry/malloc_for_haswell_hardware_transactional`. Viewed Apr. 27, 2015.

[11] D. Dice. Private Communication, Sept. 14 2014.

[12] D. Dice, T. L. Harris, A. Kogan, Y. Lev, and M. Moir. Pitfalls of lazy subscription. In *6th Workshop on the Theory of Transactional Memory (WTTM 2014)*, Paris, France, July 14 2014. `http://www.gsd.inesc-id.pt/~mcouceiro/wttm2014/html/abstracts/dice.pdf`.

[13] C. Eder and H. Schoenemann. Xmalloc, 2012. `https://github.com/ederc/xmalloc`. Viewed Nov. 13, 2014.

[14] J. Evans. A scalable concurrent `malloc(3)` implementation for FreeBSD. In *BSDCan — The Technical BSD Conference*, Ottawa, Canada, May 2006. `http://people.freebsd.org/~jasone/jemalloc/bsdcan2006/jemalloc.pdf`.

[15] J. Evans. Behavior of `madvise(MADV_FREE)`, Oct. 12 2012. `http://lists.freebsd.org/pipermail/freebsd-arch/2012-October/013287.html`. Viewed Apr. 27, 2015.

[16] J. Evans. Personal communication, Mar. 2015.

[17] Y. Feng and E. D. Berger. A locality-improving dynamic memory allocator. In *Proceedings of the 2005 Workshiop on Memory Systems Perforamnce (MSP)*, pages 68–77, Chicago, IL, June 2005. doi:10.1145/1111583.1111594.

[18] Free Software Foundation. GCC, the GNU compiler collection. `http://gcc.gnu.org`, 2014. Viewed Nov. 2, 2014.

[19] W. Gloger. Wolfram gloger's `malloc` homepage, May 2006. `www.malloc.de/en/`. Viewed Nov. 9, 2014.

[20] M. Hertz, Y. Feng, and E. D. Berger. Garbage collection without paging. In *ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation (PLDI)*, pages 143–153, Chicago, IL, June 11–15 2005. doi:10.1145/1065010.1065028.

[21] R. L. Hudson, B. Saha, A.-R. Adl-Tabatabai, and B. C. Hertzberg. McRT-malloc — a scalable transactional memory allocator. In *Proceedings of the 5th International Symposium on Memory Managment (ISMM)*, pages 74–83, Ottawa, Canada, June 2006. doi:10.1145/1133956.1133967.

[22] Intel. Intel 64 and IA-32 architectures software developer's manual — combined volumes: 1, 2a, 2b, 2c, 3a, 3b and 3c. `https://www-ssl.intel.com/content/dam/`

www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-manual-325462.pdf, June 2013.

[23] ISO/IEC. Information technology – programming languages – c. Standard 9899:2011, Dec. 8 2011.

[24] B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. Prentice Hall, Inc., second edition, 1988.

[25] D. E. Knuth. *The Art of Computer Programming*, volume 1. Addison Wesley, Reading, MA, 2nd edition, 1973.

[26] D. G. Korn and K.-P. Vo. In search of a better malloc. In *Proceedings of the Summer '85 USENIX Conference*, pages 489–506, 1985.

[27] A. Kukanov and M. J. Voss. The foundations for scalable multi-core software in Intel Threading Building Blocks. *Intel Technology Journal*, 11(4):309–322, Q4 2007. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.71.8289&rep=rep1&type=pdf.

[28] P.-Å. Larson and M. Krishnan. Memory allocation for long-running server applications. In *Proceedings of the International Symposium on Memory Management (ISMM'98)*, pages 176–185, Vancouver, Canada, Oct. 1998. doi:10.1145/286860.286880.

[29] D. Lea. A memory allocator, 1996. http://g.oswego.edu/dl/html/malloc. Viewed Nov. 3, 2014.

[30] C. Lever and D. Boreham. `malloc()` performance in a multithreaded Linux environment. In *Proceedings of FREENIX Track: USENIX Annual Technical Conference*, San Diego, CA, June 2000. https://www.usenix.org/legacy/events/usenix2000/freenix/full_papers/lever/lever.pdf.

[31] V. B. Lvin, G. Novark, E. D. Berger, and B. G. Zorn. Archipelago: Trading address space for reliability and security. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XIII)*, pages 115–124, Seattle, WA, Mar. 1–5 2008. doi:10.1145/1346281.1346296.

[32] D. J. Magenheimer, L. Peters, K. Pettis, and D. Zuras. Integer multiplication and division on the hp precision architecture. In *Proceedings of the 2nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-II)*, pages 90–99, Palo Alto, California, Oct. 5–8 1987. doi:10.1145/36206.36189.

[33] M. Matz, J. Hubička, A. Jaeger, and M. Mitchell. System V Application Binary Interface AMD64 architecture processor supplement draft version 0.99, May 2009. www.x86-64.org/documentation/abi.pdf.

[34] M. M. Michael. Scalable lock-free dynamic memory allocation. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation (PLDI)*, pages 35–46, Washington, DC, June 2004. doi:10.1145/996893.996848.

[35] S. Oboguev. The deferred set priority facility for Linux, July 2014. https://raw.githubusercontent.com/oboguev/dprio/master/dprio.txt. Viewed Apr. 27, 2015.

[36] S. Oboguev. Sched: deferred set priority (dprio), July 2014. http://lists.openwall.net/linux-kernel/2014/07/28/41. Viewed Apr. 27, 2015.

[37] F. Schanda. Bug 206 - malloc does not align memory correctly for sse capable systems. https://sourceware.org/bugzilla/show_bug.cgi?id=206, June 2004. Viewed Apr. 27, 2015.

[38] S. Schneider, C. D. Antonopoulos, and D. S. Nikolopoulos. Scalable locality-conscious multithreaded memory allocation. In *Proceedings of the 5th International Symposium on Memory Managment (ISMM)*, pages 84–94, Ottawa, Canada, June 2006. doi:10.1145/1133956.1133968.

[39] *Standard Template Library Programmers Guide – Allocators section*. SGI, Feb. 1997. Archived as https://web.archive.org/web/20010221202030/http://www.sgi.com/tech/stl/Allocators.html. Viewed Nov. 10, 2014.

[40] D. Vyukov. Possible problem in scalable allocator, Nov. 2008. https://software.intel.com/en-us/forums/topic/299796. Viewed Nov. 3, 2014.

[41] T. Yang, E. D. Berger, S. F. Kaplan, and J. E. B. Moss. CRAMM: Virtual memory support for garbage-collected applications. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 103–116, Seattle, WA, Nov. 6–8 2006. https://people.cs.umass.edu/~emery/pubs/06-25.pdf.