CSC469 Fall 2019

Assignment 3 Report

Tudor Brindus and Jason Pham
(tudor.brindus@mail.utoronto.ca, jason.pham@mail.utoronto.ca)

November 15, 2019

1 Abstract

In this report, we explore the development of a cache-aware allocator similar to Hoard[1], called a3alloc. We describe key changes to Hoard, and analyze performance against `libc` and `kheap` on 6 benchmarks: `larson`, `linux-scalability`, `cache-scratch`, `cache-thrash`, `phong`, and `threadtest`.

2 Allocator Design

At a high level, a3alloc is very similar to Hoard: it maintains per-core heaps of page-sized "superblocks" (the smallest unit of allocation) in order to service requests for memory. Threads first allocate from their core-local heap, and if it is exhausted, attempt to allocate from a "global" heap. This means that for most memory requests that can be serviced from a core-local heap, there is no costly contention for a global lock, and memory returned is likely to already be in that core's cache.

2.1 Per-Core Heaps

a3alloc's per-core heap structure is modelled after that of Hoard. Crucially, it maintain the amount of space allocated in the heap, how much is currently in use, and the superblocks protected by a lock. Spinlocks are used for synchronization, as their low overhead compared to mutexes provided sizable performance gains on all benchmarks.

```c
typedef struct heap {
  pthread_spinlock_t lock;                 // Spinlock for synchronization
  u_int8_t heap_idx;                       // Heap (core) index
  u_int32_t in_use;                        // Bytes used; u_i in Hoard
  u_int32_t alloced;                       // Bytes allocated; a_i in Hoard
  superblock_t *bins[SZ_CLASS][NUM_BINS];  // Superblocks by size and fullness
} __attribute__((aligned(64))) heap_t;
```

Superblocks are maintained in a matrix of (size class, fullness) indices. A superblock can only be used for a particular "size class" (a power of two). That is, a superblock of size class 16 can only be used to service 16-byte allocations, and not anything larger. The second index, "fullness", represents how full (in percentage terms) a superblock is. This is described in further detail in Section 2.4.

At startup, an empty heap is allocated for each processor, padded to cacheline boundaries. The space for this region is roughly one page per 8 processors. An additional "global" heap is also allocated. Superblocks are allocated later, as needed.

Threads are matched with a core heap by taking their Linux thread ID modulo the number of cores. phread IDs in Linux are consecutive, so this gives a balanced number of threads per core heap. Further improvement may result from matching core heaps based off thread affinity, but this was outside the scope of a3alloc, where the number of threads was guaranteed to be at most the number of cores for benchmarking purposes.

2.2 Superblocks

Each heap tracks a number of superblocks, each of one page length. Superblocks are arranged in a doubly-linked list per fullness group. Each superblock contains a header including a bitmap of free sub-chunks available within it.

```c
typedef struct superblock {
  pthread_spinlock_t lock;  // Spinlock for synchronization
  u_int16_t in_use;         // Bytes currently in use, excluding header
  u_int8_t bin_idx;         // Index in heap.bins[self.sz_idx]
  u_int8_t sz_idx;          // Size class, real size is 2^sz_idx
  u_int8_t heap_owner;      // The owning heap.heap_idx
  u_int8_t bitmap[64];      // Bitmap of free chunks in this superblock
  struct superblock *next;  // Next superblock in fullness group
  struct superblock *prev;  // Previous superblock in fullness group
  u_int64_t num_pages;      // How many pages are allocated to this (huge)block
} superblock_t;
```

In a3alloc, we opted for a bitmap approach to tracking free blocks due to its low memory overhead. Maintaining a secondary, per-superblock linked list of its own blocks may have allowed us to perform more intelligent things with regards to intra-superblock locality, at a significant cost to allocation speed and memory overhead.

Heap owner indices are represented as bytes, allowing for up to 256 cores to be supported by a3alloc. This has a marginal improvement in superblock header size over direct (64-bit) pointers.

2.2.1 Allocating Superblocks

When allocating from a particular thread, a3alloc first attempts to use that thread's core heap, beginning at the head of its superblock list. If `in_use` $+ 2^{\texttt{sz\_idx}} >$ `PAGE_SIZE`, then there is no space left in the superblock, and the `next` pointer is followed. If space does exist, the first unset bit in `bitmap` is claimed, set, and a pointer to the appropriate memory returned.

If there is no free superblock in the core heap, the global heap is queried the same way, and if it too is empty, a new superblock is requested (either from the operating system, or the globally free list described below).

2.2.2 Freeing Superblocks

When releasing memory, a3alloc unsets the appropriate bit in `bitmap`, then possibly moves the superblock into a different fullness group. If a superblock has fallen below its current fullness group's threshold, it is moved to a less-full group. If not, it is still moved to the front of its fullness group's superblock list, for better locality.

a3alloc also tracks a "globally free" list of superblocks. When a superblock becomes totally free, it becomes usable again for *any* size class – as per Hoard, our experiments also showed this to help with fragmentation (particularly on the Larson benchmark).

2.3 Huge Blocks

a3alloc supports "huge blocks", or allocation requests that are greater than half of `PAGE_SIZE` (in Hoard, these requests are serviced entirely by the operating system's allocator). For these, it rounds the request up to the nearest multiple of `PAGE_SIZE`, requests a superblock of that length, and sets its `num_pages` to the number of pages the huge block occupies.

When a huge block is freed, it is broken up into `num_pages` regular superblocks, and returned the the global free list. While the huge block allocation scheme does not currently attempt to coalesce globally free chunks into one chunk to satisfy requests (so freed huge block memory will never be reused for future huge block requests).

We believe this is reasonable: most large requests of memory in an application are likely to be used for long-term, persistent data structures that don't see a lot of churn. This view appears to be supported by generational garbage collectors[2], many of which perform large allocations directly on the "old" heap, skipping the short-lived minor heaps under the assumption that large allocations tend to stick around for a long time.

2.4 Fullness Groups & Partitioning

Hoard, as well as a3alloc, rely on partitioning blocks into fullness groups, ensuring that a superblock with space can be found in $\mathcal{O}(1)$ time. We chose groups $[0, 0.25)$, $[0.25, 0.5)$, $[0.5, 0.75)$, and $[0.75, 1]$. When performing an allocation, the most-full groups are tried first, in principle resulting in low fragmentation.

This choice performed well on most benchmarks – either outperforming or closely trailing `libc` – but performed abysmally on `linux-scalability`, where it was four orders of magnitude slower.

The `linux-scalability` benchmark allocates repeatedly on multiple threads, and then frees repeatedly. There is no interspersing of `malloc` and `free`; the calls come in contiguous batches. Unfortunately, this exploited the worst-case peformance of a3alloc: when the "most full" group is *totally* full, it must still be scanned for free superblocks as per the Hoard algorithm, just in case there is *some* superblock that still has space available. In `linux-scalability`, the length of the most full group linked list grew to thousands of elements, and program runtime was dominated by scanning through the list.

To restore the $\mathcal{O}(1)$ lookup property promised by Hoard, a3alloc maintains a fifth group of *totally* full superblocks, in which no further allocations can be made. This addressed the performance on `linux-scalability` (where it began outperforming `libc` by a factor of two), with modest – but strictly positive – gains on the other benchmarks.

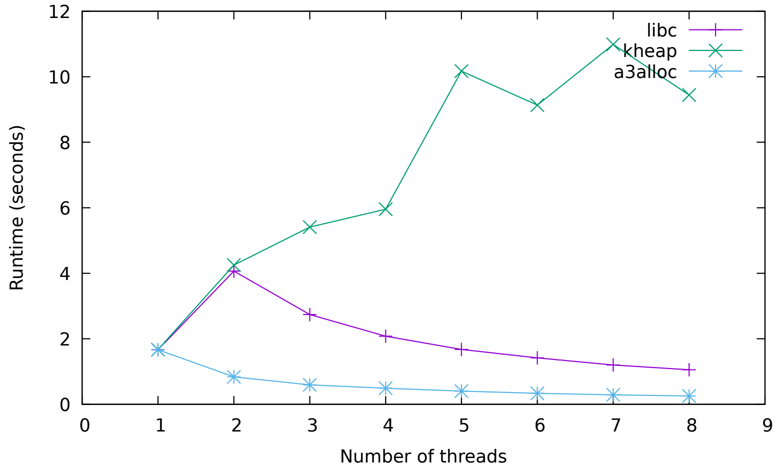
3 Benchmark Results

The performance of a3alloc against `libc` and `kheap` is demonstrated below on 6 benchmarks: `larson`, `linux-scalability`, `cache-scratch`, `cache-thrash`, `phong`, and `threadtest`. The test environment ran Linux 4.15.0, on an 12-core AMD Opteron 6344 processor operating at 1400 MHz, with 8 GB of RAM.

Our results are summarized below. Unless otherwise stated, numbers given are for the 8-thread cases.

3.1 `cache-scratch`

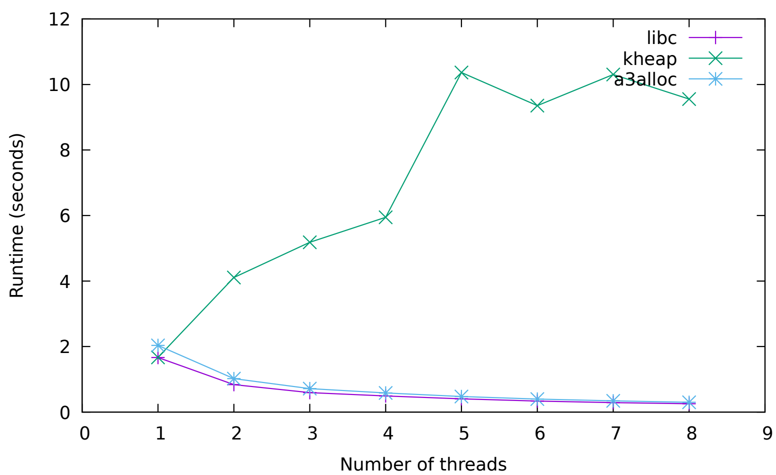
This benchmark tests passive false-sharing avoidance.



As expected – since Hoard was designed to avoid passive false-sharing – a3alloc performs well: at 8 cores, it is 5 times faster than `libc`. a3alloc also uses a constant 53 247 bytes of memory, or 325% `kheap`'s 16 383 bytes. Nonetheless, we consider this to be acceptable, given that a3alloc must maintain far more constant state than `kheap`.

3.2 `cache-thrash`

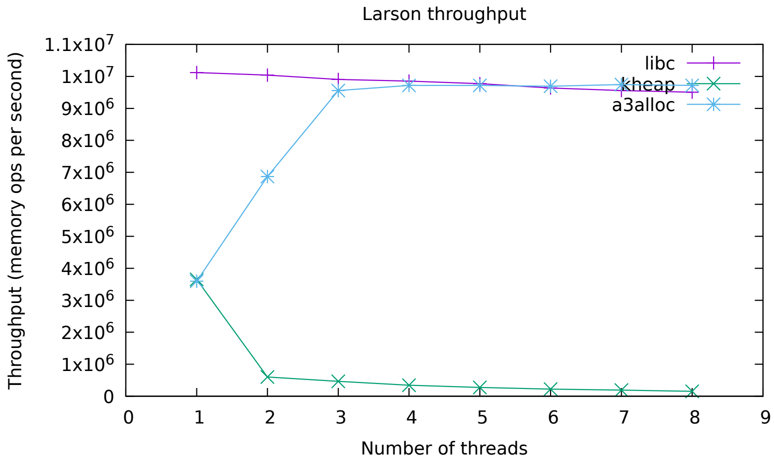
This benchmark tests active false-sharing avoidance.



Like `cache-scratch`, a3alloc performs well, but not quite as well as `libc`. We believe this is because `libc`'s allocator has a lower constant overhead when there's no lock contention. Memory usage is similar to `cache-scratch`, with a3alloc using 366% more memory.

3.3 `larson`

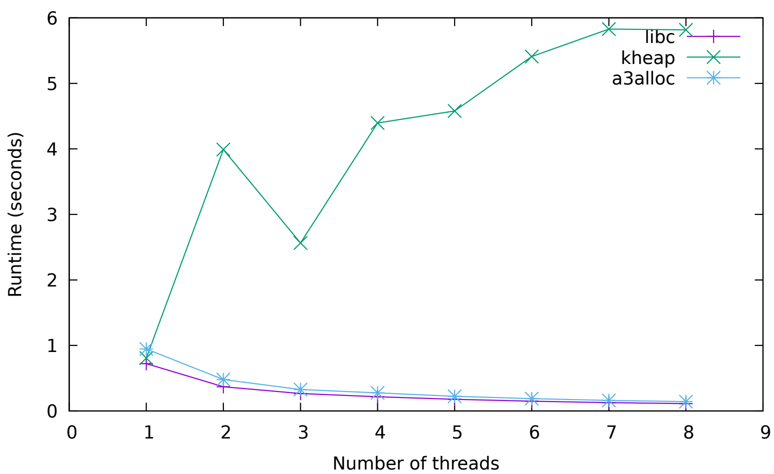
The Larson benchmark simulates a server, where each thread performs some allocations, then frees some of them on other threads.



For the most part – after a slow start of 5 threads – a3alloc begins to outperform `libc`. A server application is likely highly multithreaded, so performing poorly on small thread counts seems outweighed by performing well on high thread counts. In this benchmark, a3alloc requires 1.8 times the memory of `kheap`.

3.4 `threadtest`

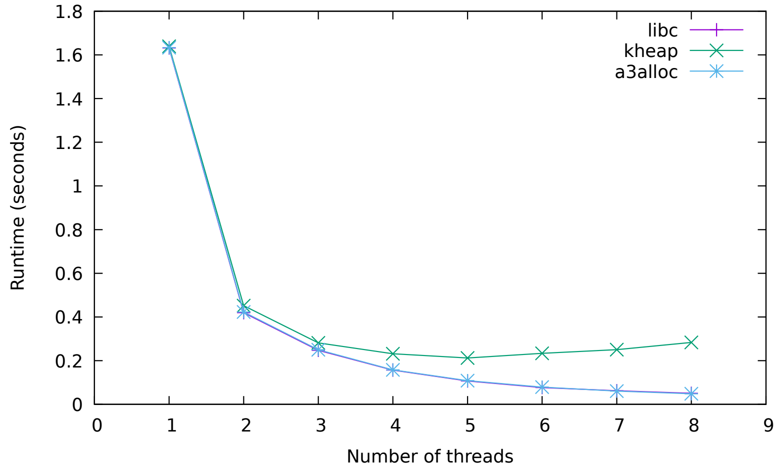
This benchmark tests allocator throughput.



a3alloc performs well, but does not outperform `libc`. Like in `cache-thrash`, we believe this to be due to a lower constant overhead in `libc`. In this benchmark, a3alloc uses 4.75 times the memory required by `kheap`, but it is still reasonable: 76 KB.

3.5 `phong`

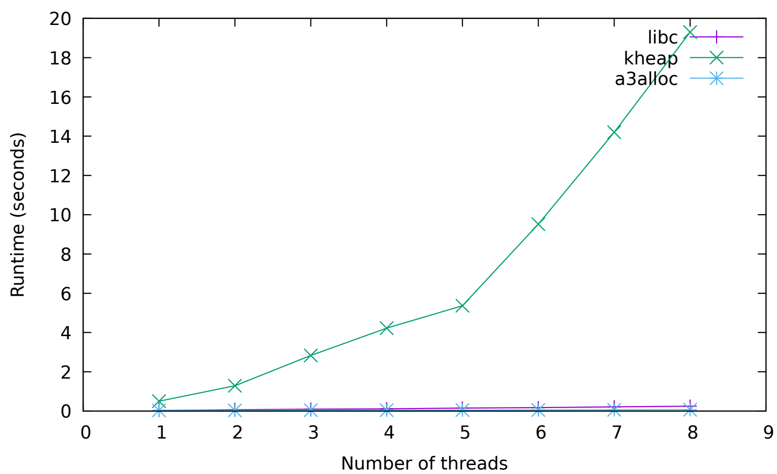
This benchmark tests allocator throughput, interspersing frees in-between allocations, to induce fragmentation.



In this test, a3alloc requires only 1.25 times the memory of `kheap` (significantly less than other benchmarks), demonstrating that a3alloc performs well against fragmentation.

3.6 `linux-scalability`

This benchmark also tests allocator throughput.



Our results were summarized in greater detail in Section 2.4, so we will omit it here. a3alloc requires 2.6 times the memory of `kheap`.

4 Addendum - Miscellaneous Optimizations

In this section, we describe several optimizations used by a3alloc for better performance.

4.1 Avoiding `gettid` Syscalls

During profiling, we discovered that on some machines, the call to `getTID` (necessary as part of the Hoard hashing scheme) would consume upwards of 30% CPU time. On others, like `wolf`, this was not the case. In order to not rely on the particulars of the machine a3alloc runs on, it computes a thread's hash once, then stores it in a thread-local storage block. This has low ($< 2\%$) impact on `wolf`, but seems to help massively on some machines.

4.2 Avoiding Integer Division

Initial profiling in `perf` showed that a large proportion of time was spent performing integer division (namely by `PAGE_SIZE`). Since `PAGE_SIZE` is guaranteed to be a power of two, we exploited this fact by maintaining its logarithm in `LOG_PAGE_SIZE`, and replacing divisions with right shifts where applicable.

We also discovered an interesting fact that GCC, at least as of version 7, does not seem to propagate power of two-ness during constant folding. The following code is an excerpt from a3alloc.

```c
static inline int to_size(int sz_idx) { return 1 << (3 + sz_idx); }

static inline int num_blocks(int sz_idx) {
  return (PAGE_SIZE - sizeof(superblock_t)) / to_size(sz_idx);
}
```

Here, despite `to_size` being (verifiably) inlined into `num_blocks`, GCC still emits a `div` instruction in `num_blocks`. It appears to not recognize that the result of `to_size` is always a power of two, and hence could be optimized into a right shift.

We addressed these situations in our code manually.

References

[1] Hoard: A Scalable Memory Allocatorfor Multithreaded Applications, `https://people.cs.umass.edu/ emery/pubs/berger-asplos2000.pdf`

[2] Real World OCaml, Chapter 21. Understanding the Garbage Collector `https://v1.realworldocaml.org/v1/en/html/understanding-the-garbage-collector.html`