CSC469 A2 Report: Implementing The Hoard Allocator

Kaiyang Wen, Ruijie Deng

November 2018

1 Introduction

In this report we will describe our implementation of scalable and parallel Hoard memory allocator [1]. The allocator is based on **sbrk(0)** with the assumption that the initial call will make the **program break** align to page boundary. Section 1 of this report will be a review of the Hoard paper; please skip this section if you are very familiar with Hoard. In second 2 section we will list the implementation difficulties we encountered as well as our solution to them. Section 3 will be our analysis of benchmark suites. In section 4 we will address potential problems and possible further progress.

1.1 Definitions

In the context of the Hoard allocator, a **block** denotes a fixed-size memory chunk. A **superblock** denotes a collection of (who has a memory chunk partitioned into) equally-sized blocks. A **heap** denotes a collection of superblocks. A **raw block** denotes a variable-size memory chunk (often largerly-sized), the allocation and deallocation of which are handled by the OS itself instead of Hoard.

K, the minimum number of superblocks within a heap, determines the threshold that a heap will start to transfer superblocks to the global heap. In our implementation, it is set to 4.

f, the emptiness threshold, determines the fraction of used space for a heap to start transferring superblocks to the global heap. In our implementation it is set to 0.25.

P, the number of processors, will be proportional to the number of heaps. In our implementation it is 8.

S, the size of a superblock, which is set equal to the page size 4096.

The **size classes** denotes the collection of power-of-two sizes, to which blocks will be sized. In our implementation, the **size classes** are `16, 32, 64, 128, 256, 512, 1024, 2048`, and each block contains an 8-byte header of meta data (which is mainly the pointer to its owner superblock), hence the actually available space for the size classes are `8, 24, 56, 120, 248, 504, 1016, 2040`. The size of a raw block is greater than 4096.

1.2 Algorithm

It is almost copied from the Hoard paper with some subtle changes (will be explained in later sections). For each superblock s, $s.u$ denotes the used space of a superblock. For each heap i, u_i denotes the used space, and a_i denotes the allocated space of a heap. Note that heap 0 denotes the global heap.

```
1 mm_malloc(sz):
2     If sz > S/2, allocate as raw memory and return.
3     i = hash(the current thread id)
4     Lock heap i.
5     Obtain the corresponding size class of sz.
```

```
6       Scan heap i's list of superblocks from most full to least.
7       If there is no superblock with free space:
8           Check the global heap.
9           If there is none,
10              Allocate S bytes as superblock s, set the owner to heap i.
11          Else,
12              Transfer the superblock s to heap i.
13              u[0] = u[0] - s.u
14              u[i] = u[i] + s.u
15              a[0] = a[0] - S
16              a[i] = a[i] + S
17      u[i] = u[i] + sz
18      s.u = s.u + sz
19      Unlock heap i.
20      Return a block from the superblock.
```

```
1   free(ptr):
2       If ptr is a raw memory, free to OS.
3       Find the superblock s this block comes from and lock it.
4       Lock heap i, the owner of s.
5       Deallocate the block from the superblock.
6       u[i] = u[i] - block size.
7       s.u = s.u - block size.
8       If i = 0, unlock heap i and the superblock and return.
9       If u[i] < a[i]-K*S and u[i] < (1-f)*a[i],
10          Transfer a mostly-empty superblock s1 to the global heap.
11          u[0] = u[0] + s.u
12          u[i] = u[i] - s.u
13          a[0] = a[0] + S, a[i] = a[i] - S
14      Unlock heap i and the superblock.
```

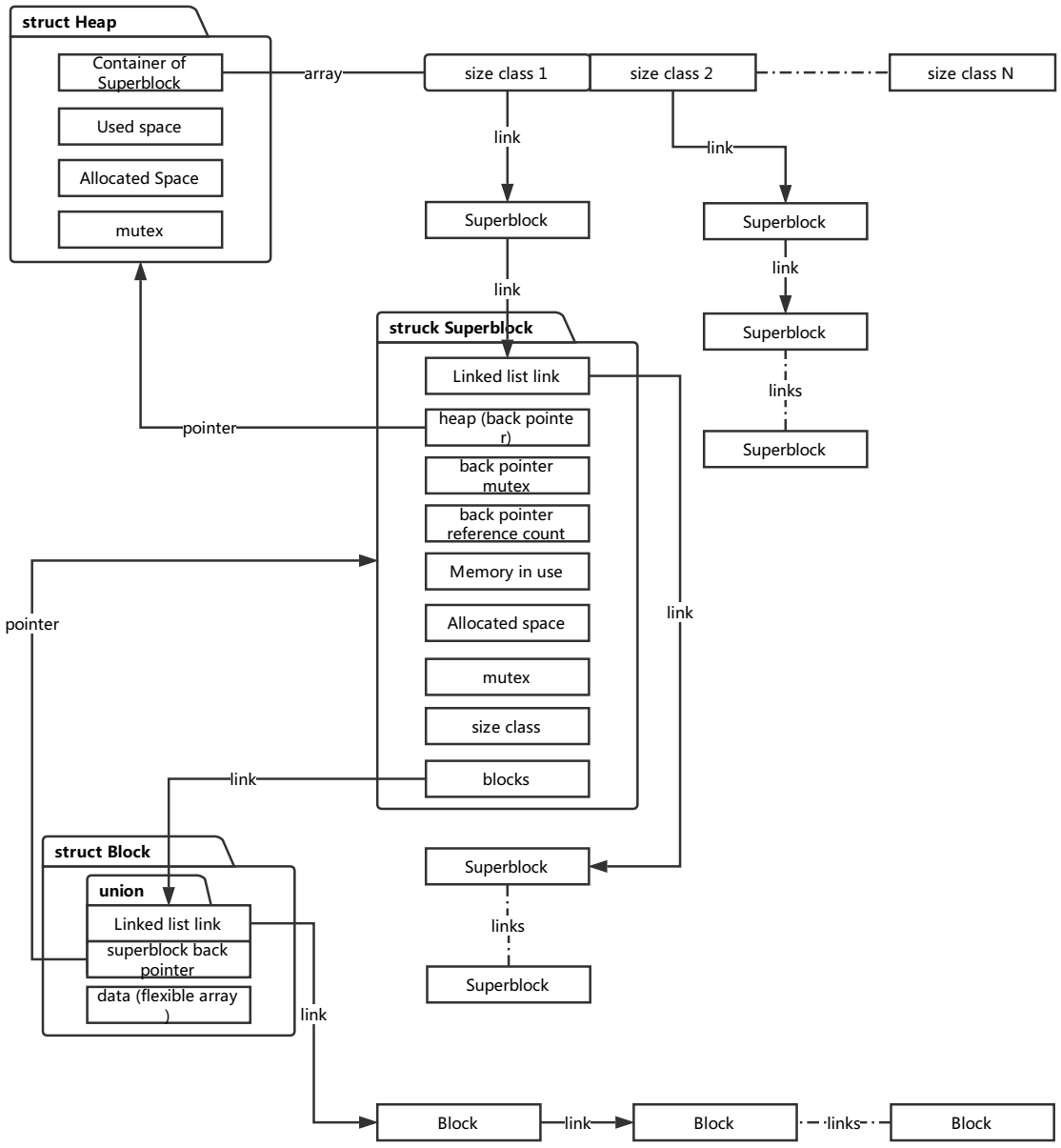
The idea behind fixed-size blocks is to avoid fragmentation caused by allocations of small memory chunks, and evenly-partitioned superblocks and multiple heaps are to reduce most passive and active false sharing.

We can see from the pseudo code, if the size of memory to be allocated or freed is greater than `S/2`, Hoard let the OS itself to do the memory allocation or deallocation.

Otherwise, for allocation, Hoard first will hash the current thread id, go to the corresponding heap `i`, and try getting a superblock with free space (of the size class) from it. If it fails, it will then try getting one from the global heap. If the global heap does not contain one either, it will go to the OS for a new superblock. The superblock acquired from outside heap `i` will be transferred to heap `i`. Finally, it will update the records for memory allocated and in use, acquire a block from the acquired superblock and return it.

For deallocation, Hoard will first find the superblock a chunk of memory comes from and lock it, and then will lock the heap this superblock is contained in. It then will deallocate the block from the superblock, and update the records for memory allocated and in use. If the superblock is not contained in the global heap, and if the heap is *too empty*, determined by `u[i] < a[i]-K*S and u[i] < (1-f)*a[i]`, it will then transfer the mostly-empty superblock `s1` to the global heap, and update the records for memory allocated and in use. Finally it will unlock the heap and the superblock and return.

2 Implementation Specifics



2.1 Structure

As we can see from the diagram, a heap structure contains an array of doubly linked list of superblocks (ordered by available space) where each linked list contains superblocks of the same size class, and a mutex for itself. A superblock structure contains a linked list pointer, a pointer to its owner heap, a mutex for the pointer to its owner heap, a reference count to this pointer as well, a mutex for itself, a linked list of blocks it stores (in LIFO order), and the size class of the blocks. Both of heaps and superblocks contain the

information for the memory allocated and in use inside themselves.

2.2 Information Update Relaxation

In our implementation, we relaxed line 17 and 18 in `malloc`. Instead of adding `sz` to `u[i]` and `s.u`, we add `block size` to `u[i]` and `s.u`.

The point of relaxation is that firstly, in `malloc` it uses `sz` to update `u[i]` and `s.u` but in `free` it uses `block size` instead. We assume that in the context of Hoard `block size` is non-smaller than but not necessarily equal to `sz`. To well define `u[i]` and `s.u`, we unify the update by `block size` for both.

On the other hand, we are convinced that using block size instead of the original `sz` will not affect the expected performance, but even tighten the performance analysis. Notice that in the inequality at line 9 of `free`, we are using `u[i] < a[i]-K*S and u[i] < (1-f)*a[i]` to tell if heap `i` is *too empty*. If we use `sz` to update, we got `s.u = sum of sz of blocks in s`. Since we are using size classes, the memory that are actually occupied is non-smaller than, and in worst case almost doubly greater than `s.u`. This does not really help to decide if a superblock is empty or not, as if a superblock `s` contains the same number of much *fuller* blocks as (than) the other superblock `t` does, `s.u` will be significantly greater than `t.u`, but actually in the sense of *emptiness* they are equally qualified since they have same amount of free memory left. However, if `block size` is used instead, `s.u` will equal `t.u` and this implies the exact fact that they are *equally empty*.

Aside, the value of `sz` is different for each allocation, so we need to store this value for each block, meaning that the cost of allocation will become higher.

2.3 Handling "Larger" Raw Blocks

The Hoard paper did not specify anything about the raw blocks. In our implementation, we simply create a linked list of not-in-use (freed) raw blocks ordered by their size. As we are allocating any new memory, we first check this linked list to find one freed raw block with most fit size, remove it from the list and use it as the allocation result. If the found raw block's size is too much greater than required (in our implementation, *too much greater* is more than the page size), we divide the found raw block into 2, take the one whose size fits for requirement as the allocation result, and insert what is left back into the ordered linked list. If there is no freed raw block in the linked list with non-smaller size than required, we simply call `mem_sbrk` for a new raw block as the result. Also, apparently when we are freeing a raw block, we simply put it into the ordered linked list.

Furthermore, when we are calling `mem_sbrk` to require space from the heap area, we only require *k * page size* bytes where `k` is a positive integer. Thus by doing this the raw blocks will be sized to multiple times of page size, and when we are allocating superblocks, we can find (and then possibly divide) a raw block from the linked list of not-in-use raw blocks. We assume that there are much more allocation of small memory chunks (which is often the fact in most cases), hence most allocations are for superblocks, not for raw blocks. This helps us to reduce fragmentation, because it allows more superblock allocations from freed raw blocks. The extra fragmentation wasted within raw block is considered minor due to the assumption that large allocation does not happen so often. This extra fragmentation has an upper bound of 1 times the page size per raw block.

2.4 Finding the Owner of Superblocks

In our implementation, there is a back pointer to the owner heap of a superblock. We found that this pointer is shared across malloc and free, and is modified concurrently in line 12 of `malloc` and line 10 of `free`. The point of having this pointer is to speed up line 4 of `free`; note that without this pointer, the code for finding the owner of a superblock will be a while loop acquiring the lock of each heap and then performing lookup inside (That solution requires one to lock all heaps in the worst case; if we do not lock all the visited heaps, in the

worst case the superblock can move around by another thread, which may lead to starvation during the lookup). But with this pointer, there is a very hidden concurrency issue. We don't expect the value of this pointer to be changed before we take the lock of the owner heap.

A solution is to serialize all transfer operation using a mutex (in the diagram it is called the *back pointer mutex*), so at any time line 4 of `free` is executed, the correct parent heap pointer will be read. And in order to guaranteed that the owner of a lock is not changed inside line 4 of `free`, we need to have a reference count for each superblock, and forbid superblock transfers if the count is nonzero.

In our code for line 4 of `free`, we first acquire the *back pointer mutex*, increment the reference count, read the pointer value, and release that mutex before we try to lock the heap. After we acquire the lock for the owner heap, we will decrement this reference count.

This solution adds complexity to the code, but we believe it is one of the simplest **correct** solutions. We tried to serialize line 4 of `free` using a mutex as well, but found that there is mutal dependency between the the heap's mutex and the pointer's mutex.

The negative effect of this solution is that it would cause performance degradation, because we will need to take an additional lock each time we are attempting to transfer a superblock; the cost of acquiring lock is naturally high.

Also, by forbidding certain superblocks' movement, we have the chance of not transferring the most empty superblock into another heap, meaning that the amount of fragmentation will become randomly worse.

2.5 Just-In-Time Meta-Data Creation

In order to reduce the number of cache invalidation, we are maintaining a pointer to the last (in the sense of memory address) unused block within a superblock. Initially we do not put the blocks inside the free blocks linked list; we will only add the blocks to the linked list inside the free function. Since blocks are *sparsely* located, creating a meta-datum for each block will guarantee to invalidate one cache line.

If we put all the blocks into the linked list at the beginning, almost certainly the N cache lines will be updated (where N denotes the number of blocks in a superblock), and worse, those updated cache lines may not necessarily be used at all.

2.6 Design Flaw of the Block Structure

When we run the Larson test suite, we found our implementation does not scale well. After careful inspection, we discover that our implementation introduces many false sharing. We should not put the linked list data at the beginning of each block. More on this (including a fix we proposed) in the next section.

3 Benchmarking

The parameters are tuned based on the hardware specification of teaching lab timing server [2]. The server has a AMD Opteron(tm) Processor 6344 model CPU, 8 cores with 1400MHz main clock frequency for each core, 2MB cache size, and 64B cache alignment. Based on the *cpuinfo*, the page size is implied to be 4KB. This section will only discuss the results on the timing server.

3.1 Sequential Case

3.1.1 Fragmentation

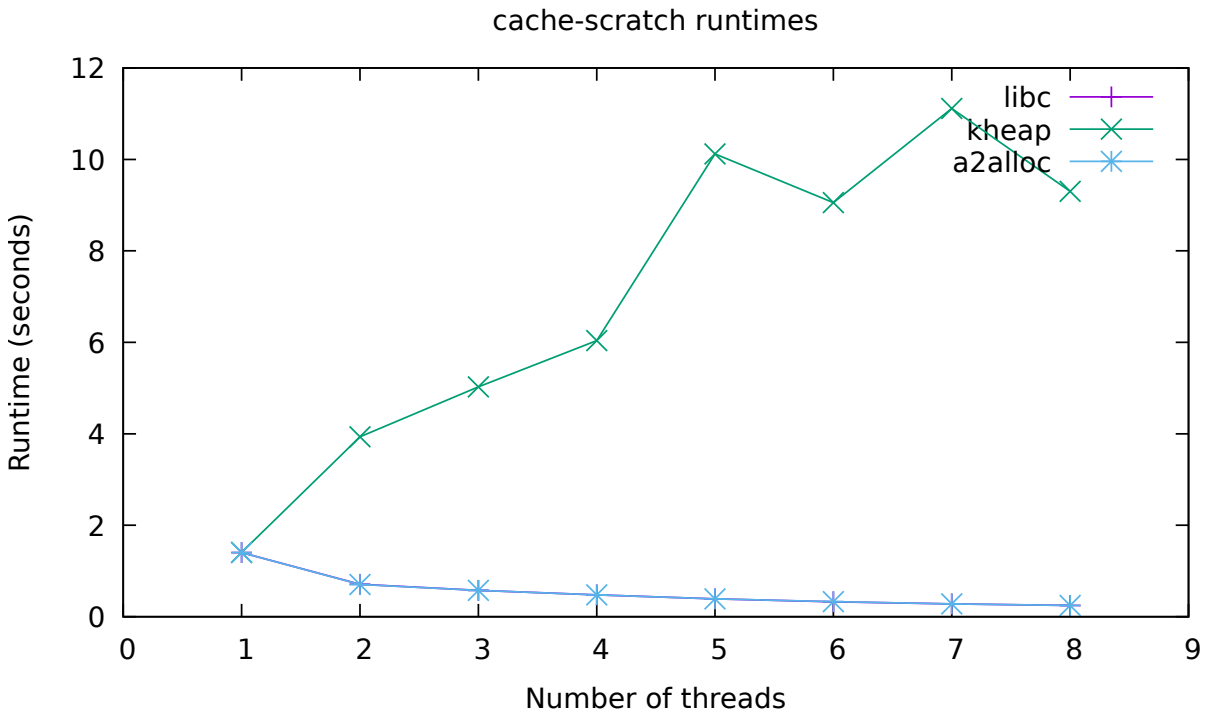
$$\frac{1}{N}\sum_{\text{Benchmark}}\frac{\text{Memory Used}_{\text{a2alloc}}}{\text{Memory Used}_{\text{kheap}}} = \left(\frac{20479}{12287} + \frac{16383}{12287} + \frac{27631615}{13709311} + \frac{2023423}{1990655} + \frac{24575}{12287}\right) = \text{approx. } 1.61$$

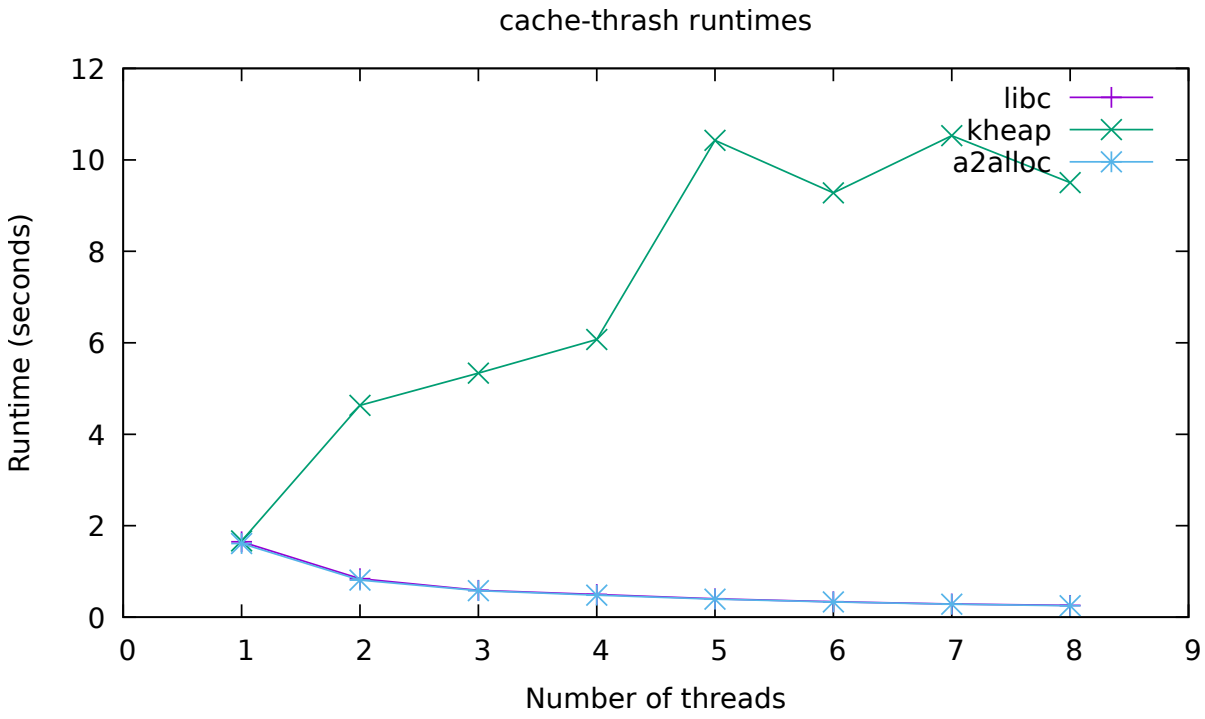
Note: we excluded the data from Larson test case. Associating heaps for each processor certainly increases fragmentation. It is unavoidable.

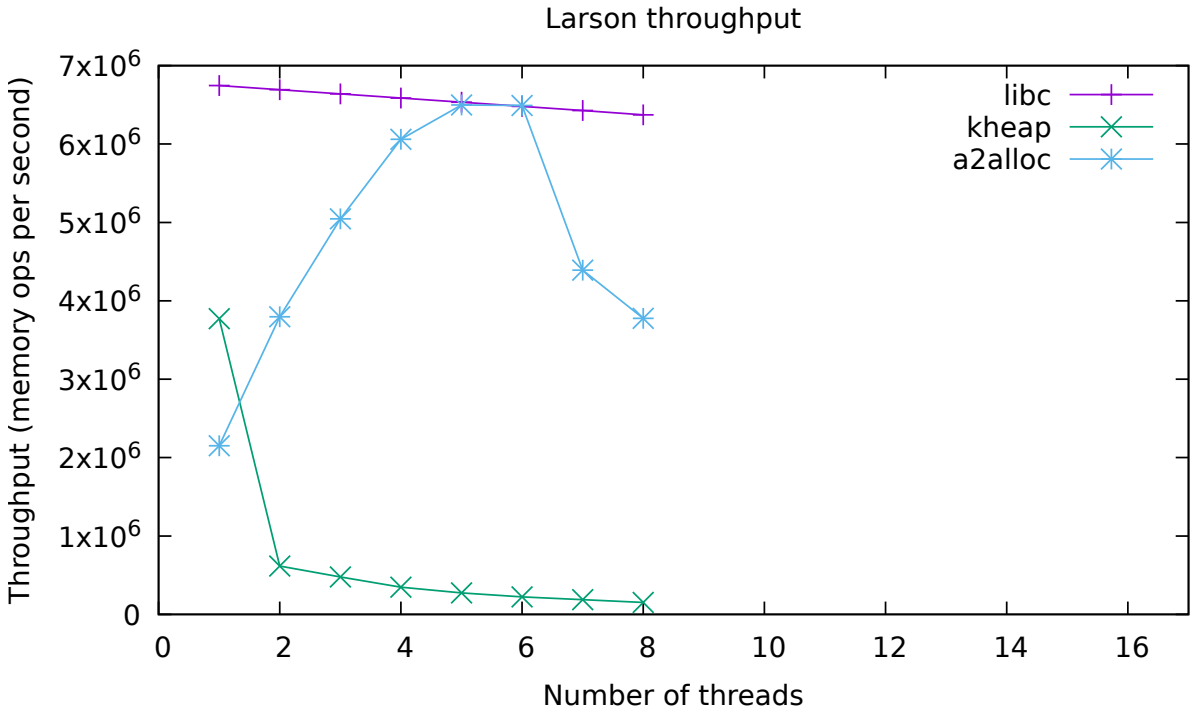
3.1.2 Runtime

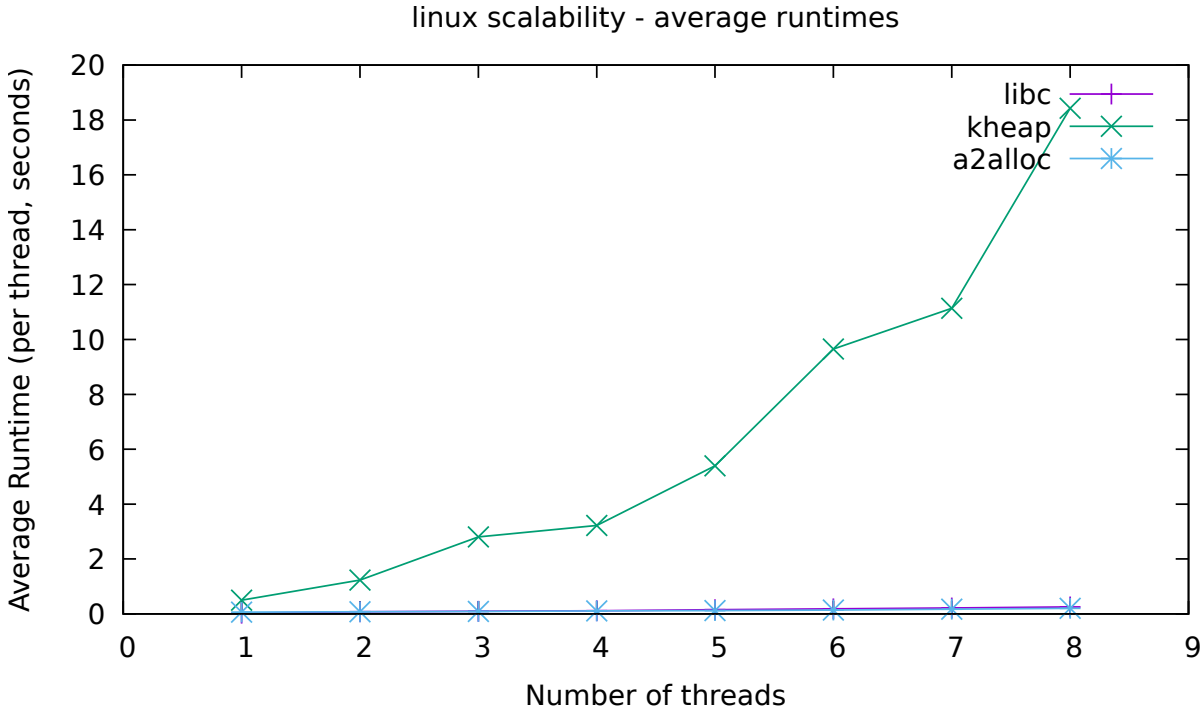
$$\frac{1}{N}\sum_{\text{Benchmark}}\frac{T_{\text{libc}}}{T_{\text{a2alloc}}} = \frac{1.409460}{1.400995} + \frac{1.649129}{1.612911} + \frac{1/6746321}{1/2156244} + \frac{0.027469}{0.065815} + \frac{2.135053}{2.177003} + \frac{0.839412}{2.151459} = \text{approx. } 0.689$$

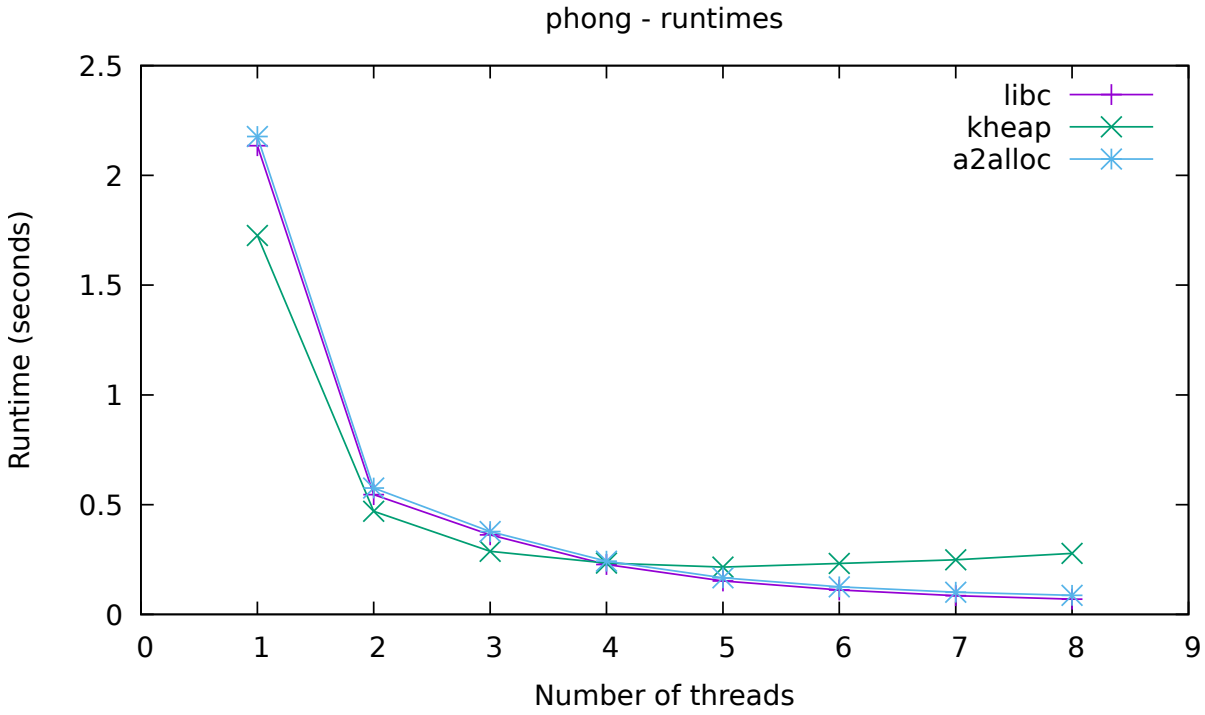
3.2 Parallel Case

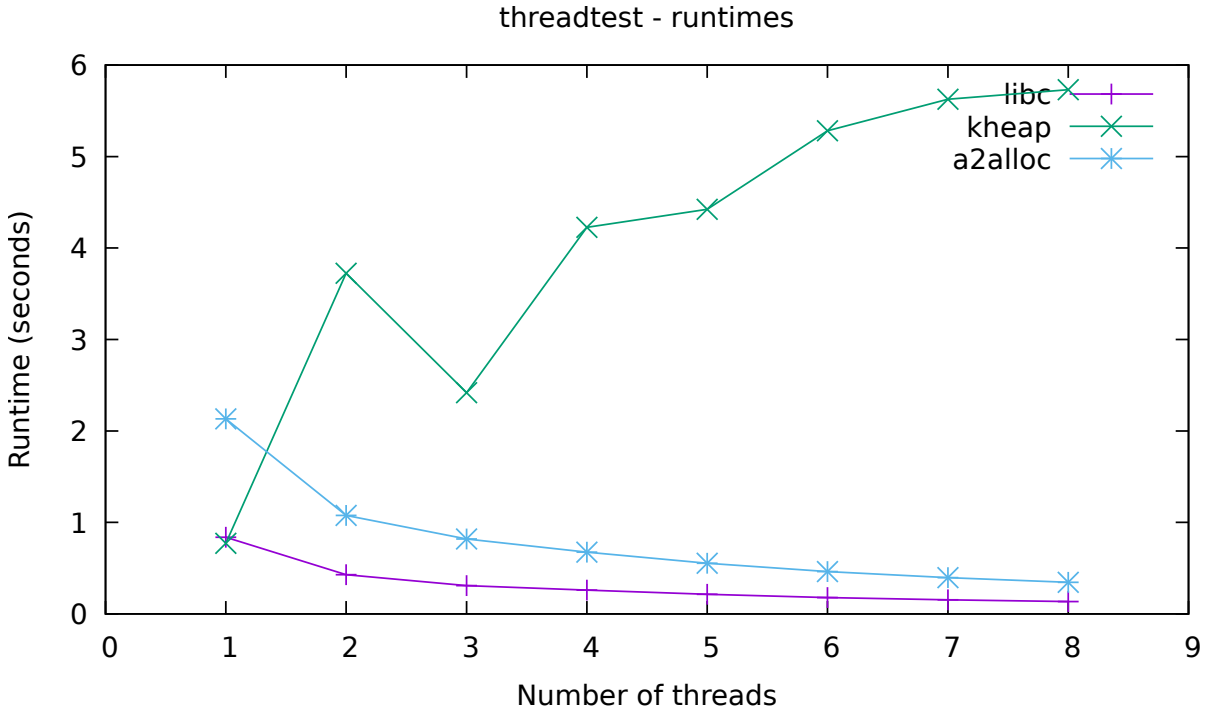





Larson throughput
Throughput (memory ops per second)
7x10^6
6x10^6
5x10^6
4x10^6
3x10^6
2x10^6
1x10^6
0
0
2
4
6
8
10
12
14
16
Number of threads
libc
kheap
a2alloc


linux scalability - average runtimes
Average Runtime (per thread, seconds)
20
18
16
14
12
10
8
6
4
2
0
0
1
2
3
4
5
6
7
8
9
Number of threads
libc
kheap
a2alloc





In terms of runtime, except for the *threadtest* and *Larson* test suites, our results are matching the ones of Linux *libc's* implementation.

3.2.1 Analysis of Threadtest

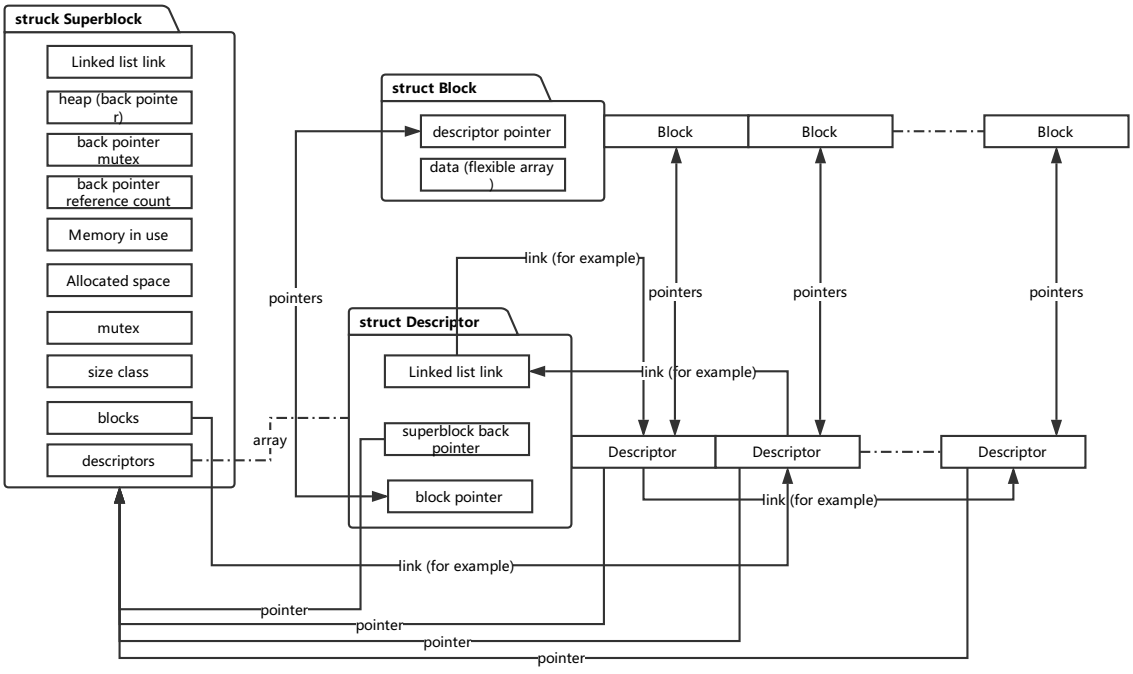
For the *threadtest*, our result is almost a multiple of the runtime of libc's implementation. Since the test suite is allocating the same amount of memory, we believe the extra work we have done here is related to the lock acquisition during free.

3.2.2 Analysis of Larson

The difference in *Larson* test suite hints that our implementation is weak against passive false sharing. It is not obvious at all, but we finally discovered that the performance degregation actually comes from our design flaw instead of from the Hoard allocator algorithm.

In each free operation we will link the blocks back to the free blocks linked list. **Unfortunately, this is done in another thread other than the allocator of that chunk of memory**. Since the Larson test suite is simulating a server-client application, and the "client" is not modifying the shared memory, then each free operation will invalidate one **different** cache line on the "server" (assuming clients and server run on different cores of the CPU), because the allocator is makeing changes to the block's meta-datum.

An easy fix is to reduce the number of cache line invalidations. If all linked list meta-data are stored contiguously, then in the best case, only a typically small number of cache lines will be invalidated (versus the cache lines containing those blocks). A diagram of the (corrected) design is attached below.



Unfortunately, due to due date, we don't have such implantation.

4 Conclusion

We believe the performance bottleneck of our implementation is in finding the the owner of a superblock, since nowadays the cost for acquiring lock is considerably high. The spin lock inside a superblock also

increases fragmentation because the movement of superblock is prohibited when the superblock is referenced by *free*, which forces the allocator to request additional system memory.

We also did some research to another scalable and parallel allocator, *JEMALLOC* [3] (the allocator on Android phone), and found that the size classes can be logarithmicly spaced in order to further reduce fragmentation. (We did not implement such function due to complexity).

Finally, since Hoard is a really large scale allocator and our benchmark suite is only testing 8 threads at maximum, we are not really testing the emergent condition (for example, a data base server on an 8 core machine may have more than 200 worker threads). In theory, the Hoard allocator should work fine on multi-threading applications (, since the Hoard allocator is hashing the thread id instead of the processor id).

References

[1] Emery D. Berger, Kathryn S. McKinley, Robert D. Blumofe, Paul R. Wilson. *Hoard: A Scalable Memory Allocator for Multithreaded Applications.* ASPLOS IX, 2000.

[2] CPUINFO of the test machine: `https://www.teach.cs.toronto.edu//~csc469h/fall/assignments/a2/timing_server_cpuinfo.txt`

[3] JEMALLOC, http://jemalloc.net/jemalloc.3.html