CSC469
ASSIGNMENT 2

DANIEL BLOEMENDAL
SIMON SCOTT

Contents

Date: November 15, 2013.

1. Design

1.1. **Overview.** Our allocator closely follows the Hoard design discussed in the Emery Berger et al. paper. Memory is reserved in page sized pieces in the form of superblocks. These superblocks reside in a number of thread local heaps. The number of these heaps is chosen to be some multiple of the number of processors on the system. Superblocks are further divided into size classes, which we chose to be spaced apart by powers of 2. Furthermore, the superblocks are stored into fullness groups based on how much free space exists in a given superblock. Finally, a global heap is used to prevent unbounded memory use in certain scenarios. When superblocks reach some emptiness threshold they are moved to the global heap, and can be acquired by any thread.

1.2. **False sharing avoidance.** The primary goal of the Hoard allocator is to avoid false sharing. Our allocator achieves this by using thread local heaps. To begin with, the superblocks were carefully chosen to be aligned by cache line boundaries, as seen in listing 1.

Listing 1. Superblock & allocation structure

```
// Superblock structure
struct SUPERBLOCK_T {
    heap_t*         heap;
    uint8_t         group;
    int             size_class;
    size_t          block_size;
    size_t          block_count;
    size_t          block_used;
    blockptr_t      next_block;
    blockptr_t      next_free;
    superblock_t*   prev;
    superblock_t*   next;
};

// Allocation structure
struct ALLOCATION_T {
    int type;
    union {
        superblock_t sb;
        largeblock_t lb;
    };
} __attribute__((aligned(ARCH_CACHE_ALIGNMENT)));
```

This guarantees that if two threads allocate memory on their own superblocks no false sharing should occur. However, it should be noted that the process of placing nearly empty superblocks into the global heap, to be acquired by arbitrary threads, does allow for some limited amount of false sharing to occur. However, in practice since these superblocks are mostly empty once acquired the likelihood of false sharing is negligibly small.

1.3. **Unbounded memory use.** The global heap is used as a way to avoid certain scenarios in which unbounded memory usage may occur. One such scenario is a producer consumer relationship. A producer thread may produce some object and hand it off to the consumer thread. Once the consumer thread is done with the object, if there is no way for the producer to reclaim the memory, this can lead to unbounded memory usage. In our allocator, the consumer would end

up relinquishing the superblock after freeing the object, assuming the superblock is mostly empty afterwards. This successfully mitigates the unbounded memory problem.

1.4. **Fragmentation.** Superblocks belong to a set of *size classes*, where a superblock of a given size class only has blocks of the given size. In our implementation these classes are powers of 2. This limits internal fragmentation to a factor of 2 (Berger et. el., p.2) [1].

1.5. **Cache locality.** Freed blocks are placed in a LIFO. This increase the likelihood of reallocating recently used memory. This will take advantage of resident TLB entries and other cache effects.

1.6. **Synchronization.** In our implementation, locking is done at thread local heap level. This seems to work very well if threads mostly allocate and free their own memory only. However, there are some rare scenarios where significant lock contention occurs. This is illustrated in the Larson benchmark. If threads often free memory allocated by other threads they prevent other threads from making progress while the deallocation occurs.

2. Benchmarking

2.1. **Hypothesis.** We expected to see very good results in the cache tests. After all, the primary focus of the Hoard design is to avoid false sharing. Since there were no significantly computationally expensive algorithms in the code base, we also expected to see reasonably good speed. We also expected to see less impressive results in the Larson test due to potentially high lock contention.

2.2. **Hardware.** The data was collected in Arch Linux running in VMware Workstation on top of the following hardware 1.

Figure 1. vmbox.home hardware

Host	vmbox.home
CPU	Intel® Core™ i7-2600K 3.40GHz clock 8MB cache 4 cores 8 threads
Memory	16GB physical 2GB swap
Kernel	3.11.6-1-ARCH x86_64

2.3. **Throughput.** As expected, the allocator did well in the cache tests. The figures below show our allocator to be on par with the Linux system's standard C library.

FIGURE 2. Passive false sharing

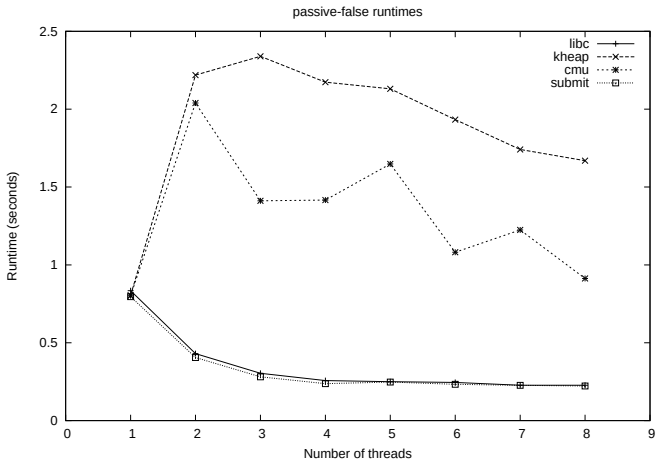
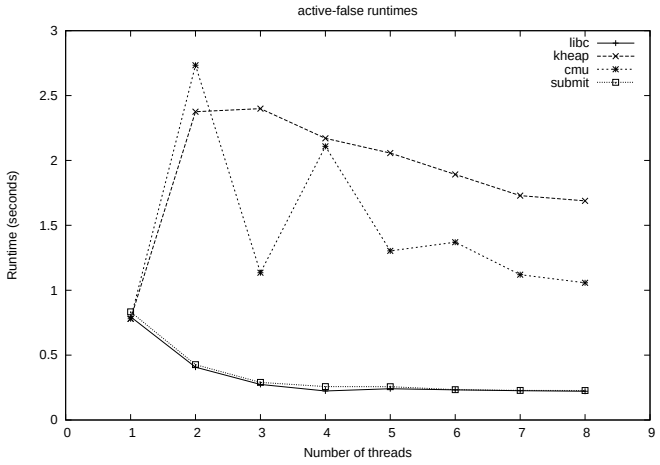


FIGURE 3. Active false sharing



The allocator did not do as well in Larson and the Threadtest. There also seems to be some type of bottleneck at around 8 threads. This is seen in the standard C allocator in the Threadtest benchmark, but is more pronounced in our allocator in both Larson and the Threadtest. This is most likely due to the fact that the underlying system is not really an 8 core system but rather a hyperthreaded 4 core system. Hyperthreading only accelerates context switches.

FIGURE 4. Larson

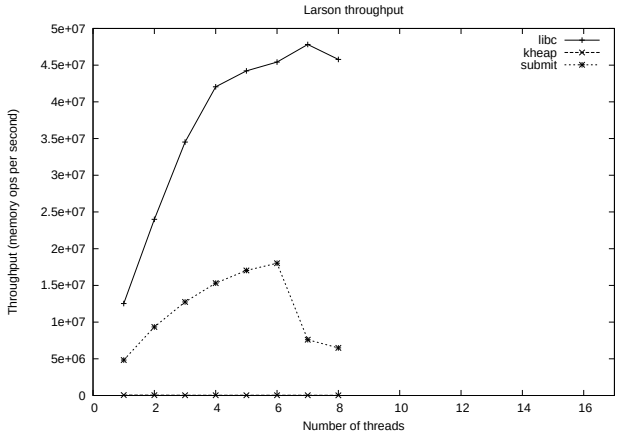
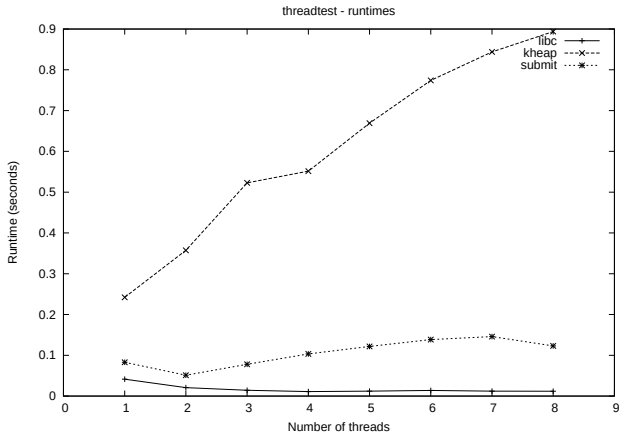
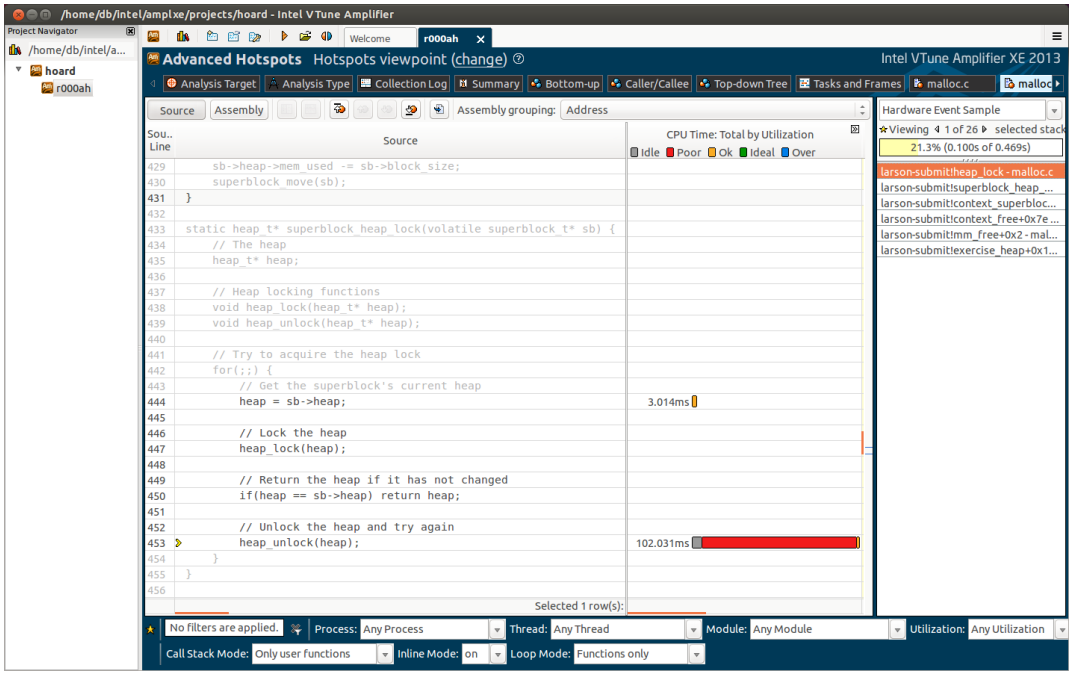


FIGURE 5. Threadtest



2.4. **Lock contention analysis.** To confirm our worst fears about the Larson test we decided to perform a more detailed analysis. We used Intel® VTune™ Amplifier XE 2013 to analyze the lock contention while the Larson test is running. As mentioned earlier, we expected to see lock contention caused by the calls to free memory not allocated by the thread doing the freeing.

FIGURE 6. VTune



As expected lock contention is somewhat of a problem during the Larson test.

2.5. **Fragmentation.** The fragmentation results are as seen in figure 7.

FIGURE 7. Fragmentation results

Benchmark	Allocator			
	submission	libc	kheap	cmu
cache-scratch	0.169	0.030	0.109	1.000
cache-thrash	0.182	0.030	0.114	1.000
larson	0.395	0.592	0.039	SEGFAULT
threadtest	0.345	0.091	0.004	SEGFAULT

With larger workloads present in *larson* and *threadtest* our allocator does reasonably well relative to the other allocators tested. It beats *libc* in *threadtest* and comes close to *libc* in *larson*.

However, we claim that the fragmentation score is biased in the case of *cache-scratch* and *cache-thrash*, which both have small workloads. These two tests allocate very little and assume that 4KB is reasonable amount of memory to expect an allocator to need during operation. This unfairly penalizes Hoard in two ways. First, Hoard requires 2KB for its static structures alone. This is assuming 4 fullness groups, 9 size classes, 64 bit (8 byte) pointers and 8 processors. We have $4 * 9 * 8 * 8 = 2304$ or about 2KB. Second, Hoard and allocators like it allocate in superblocks and require at least $K * S$ worth of memory on each heap, where S is the size of a superblock and K is some constant. We note that S is usually at least page sized for good performance. Assuming 4KB pages, if even one page was allocated at this point we would be at 6KB, which is 2KB over the 4KB estimate. However, for Hoard to obtain good performance K must be greater than 0. If it is not, Hoard incurs significant performance penalties due to lock contention associated with transferring back and forth between the global and local heaps.

So now with $K > 0$, Hoard must allocate usually 1 page per processor worth of memory at a minimum. On an 8 processor system, that amounts to 32KB of memory! We are now at more than 38KB of memory, well over the suggested 4KB. Yet it is not fragmentation, since internally there is plenty of free memory to go around. This would be no problem with reasonably sized work loads, as the other benchmarks suggest. This is why we claim that the fragmentation score in *cache-scratch* and *cache-thrash* unfairly penalizes Hoard and allocators like it.

REFERENCES

[1] Emery D. Berger, Kathryn S. McKinley, Robert D. Blumofe, and Paul R. Wilson. Hoard: a scalable memory allocator for multithreaded applications. 28(5):117–128, December 2000.

E-mail address: d.bloemendal@utoronto.ca