Hoard: A Fast, Scalable, and Memory-Efficient Allocator for Shared-Memory Multiprocessors

Emery D. Berger Robert D. Blumofe
{emery,rdb}@cs.utexas.edu [*]

Department of Computer Sciences
The University of Texas at Austin
Austin, TX 78712

Abstract

In this paper, we present Hoard, a memory allocator for shared-memory multiprocessors. We prove that its worst-case memory fragmentation is asymptotically equivalent to that of an optimal uniprocessor allocator. We present experiments that demonstrate its speed and scalability.

1 Introduction

Researchers and programmers have long observed the *heap contention* phenomenon: multithreaded programs that perform dynamic memory allocation do not scale because the heap is a bottleneck. When multiple threads simultaneously allocate or deallocate memory from the heap, they will be serialized while waiting for the heap lock. Programs making intensive use of the heap actually slow down as the number of processors increases.

In terms of contention and memory consumption, there are two extreme types of allocators. A *monolithic* allocator has one heap protected by a lock and does not waste any memory. Memory freed by one processor is always available for re-use by any other processor, but every allocation and deallocation must acquire the heap lock. In a *pure private-heaps* allocator, each processor has its own private heap that is refilled as needed with large blocks of memory from the operating system. When a processor allocates memory, it takes it from its own heap. When a processor frees memory, it puts it on its own heap. Because no processor ever accesses another processor's heap, heap access is contention-free except during refills.

However, a pure private-heaps allocator can exhibit *unbounded* memory consumption for a fixed amount of memory requested. For example, consider a program in which a producer thread repeatedly allocates a block of memory and gives it to a consumer thread who frees it. If we link this program with a monolithic allocator, the memory freed by the consumer thread is re-used by the producer thread, so only one block of memory is allocated. But if we link

[*]This research was supported in part by the Defense Advanced Research Projects Agency (DARPA) under Grant F30602-97-1-0150 from the U.S. Air Force Research Laboratory. Multiprocessor computing facilities were provided by a generous donation from Sun Microsystems, Inc. In addition, Emery Berger was supported in part by a Novell Corporation Fellowship.

with a pure private-heaps allocator, the memory freed by the consumer is unavailable to the producer. The longer the program runs, the more memory it consumes.

In this paper, we present *Hoard*, an allocator for shared-memory multiprocessors that combines the best features of monolithic and pure-private heaps allocators. We prove that Hoard is *memory-efficient*. Its worst-case memory fragmentation is asymptotically equivalent to that of an optimal uniprocessor allocator. Specifically, we show that for U bytes of memory requested, Hoard allocates no more than $O(\log(M/m)\ U)$ bytes, where M and m are respectively the largest and smallest blocks requested. This bound matches the lower bound for worst-case fragmentation that holds for uniprocessor allocators [Rob77]. We demonstrate Hoard's speed and scalability empirically, using synthetic benchmarks and applications. We show that Hoard is nearly as fast as a uniprocessor allocator and that it scales linearly with the number of processors, like a pure-private heaps allocator.

The rest of this paper is organized as follows. In Section 2, we give an overview of Hoard and describe its algorithms in detail. We prove Hoard's memory efficiency in Section 3. Section 4 explains how Hoard achieves speed and scalability, which we demonstrate empirically in Section 5. We discuss previous work in the area in Section 6, and conclude in Section 7.

2 The Hoard Allocator

The overall strategy of the Hoard allocator is to avoid contention by using a local heap for each processor, like private-heaps, while bounding memory fragmentation by periodically returning memory to a globally accessible heap. Each processor has a distinct heap for each *size class* (a range of block sizes). Hoard allocates memory from the operating system in "superblocks" of S bytes that it subdivides into blocks in the same size class to avoid external fragmentation within superblocks. Hoard manages blocks larger than S separately from superblocks. When Hoard frees a large block, it is immediately placed on the globally-accessible heap for re-use by any other processor.

Each heap contains a number of superblocks. A processor allocates blocks only from superblocks on its own heap, although it can deallocate blocks from any superblock. A per-processor heap is allowed to hold no more than $2S$ free bytes. When the free bytes on a per-processor heap exceed $2S$, a superblock with free space is transferred from the per-processor heap to the globally-accessible heap. Maintaining this invariant for each heap achieves memory efficiency while keeping contention low. In the rest of this section, we provide a detailed description of Hoard's allocation and deallocation algorithms.

There are three tunable system parameters that control Hoard's behavior. S is the size in bytes of a superblock. α is the largest alignment in bytes required for a given platform. B is the base (> 1) of the exponential that determines size classes: a block of size s is in the smallest size class $c \geq 0$ such that $s \leq \alpha B^c$. In the experiments cited below, the alignment α is 8, the size of a superblock S is $32K$ and the base of the exponential for size classes B is 1.2. The SPARC architecture dictates our choice of α. We choose S to be a multiple of the system page size ($8K$) large enough to make Hoard run as fast as a uniprocessor allocator. By keeping B relatively small, we minimize the internal fragmentation caused by rounding to the nearest size class.

We number the heaps 0 to P. Heap 0 is the *process heap* accessible by every processor, while the other heaps are the *processor heaps*; processor i uses heap i. To allow us to keep track of memory consumption, Hoard maintains a pair of statistics for every size class in heap i: u_i, the number of bytes in use in heap i, and a_i, the total number of bytes held in heap i.

2.1 Allocation

The algorithm for allocation is presented in Figure 1. When processor i calls `malloc`, Hoard locks heap i and checks it to see if there is any memory available. If not, it checks heap 0 for a superblock. If there is one, Hoard transfers it to heap i, incrementing u_i by $s.u$, the number of bytes in use in the superblock, and incrementing a_i by $s.a$, the total number of bytes in the superblock. If there are no superblocks in either heap i or heap 0, Hoard allocates a new superblock of at least S bytes and inserts it into heap i (and updates a_i). Hoard then chooses a single block from a superblock with free space, marks it as allocated, and returns a pointer to that block.

```
malloc (sz)
1.  $i \leftarrow$ the current processor.
2.  Scan heap $i$'s list of superblocks
        (for the size class corresponding to sz).
3.  If there is no superblock with free space,
4.      Check heap 0 for a superblock.
5.      If there is none,
6.          Allocate $\max\{\text{sz}, S\}$ bytes
              as superblock $s$
              and set the owner to heap $i$.
7.          $a_i \leftarrow a_i + s.a$.
8.      Else,
9.          Transfer the superblock $s$ to heap $i$.
10.         $u_0 \leftarrow u_0 - s.u$
11.         $u_i \leftarrow u_i + s.u$
12.         $a_0 \leftarrow a_0 - s.a$
13.         $a_i \leftarrow a_i + s.a$
14. $u_i \leftarrow u_i + \text{sz}$.
15. Return a block from the superblock.
```

Figure 1: Pseudocode for Hoard's `malloc`.

2.2 Deallocation

The algorithm for deallocation is presented in Figure 2. Each superblock is associated with its "owner" (the processor whose heap it's in). When a processor frees a block, Hoard finds its superblock (by a pointer dereference) and marks the block as available. Hoard then locks the owner heap i and decrements u_i. (If this block is "large" (size $> S$), we immediately transfer its superblock to the process heap.) If the amount of free memory ($a_i - u_i$) exceeds $2S$, Hoard transfers its emptiest superblock to the process heap (lines 11-14).

We now show that Hoard maintains the invariant that for each size class, no processor heap i contains more than $2S$ free bytes.

Invariant: $a_i - u_i \leq 2S$

A processor calling `malloc` either decreases the amount of free memory ($a_i - u_i$) by incrementing u_i (by allocating a block from one of its superblocks, line 14), or it changes the amount of free memory from 0 to no more than S by transferring a superblock from the process heap (line 9) or allocating a new superblock (line 6). When a processor calls `free`, it increases the amount of freed memory on its heap by one block, but if the amount of free memory on its heap exceeds $2S$, it transfers the emptiest superblock to the process heap (lines 11-14). This reduces the amount of free memory by at least one block, thus restoring the invariant.

```
free (ptr)
1.  Find the superblock $s$ this block comes from.
2.  Deallocate the block from the superblock.
3.  $i \leftarrow$ the superblock's owner.
4.  $u_i \leftarrow u_i -$ block size.
5.  If $i = 0$, return.
6.  If the block is "large",
7.      Transfer the superblock to heap 0.
8.      $u_0 \leftarrow u_0 + s.u$, $u_i \leftarrow u_i - s.u$
9.      $a_0 \leftarrow a_0 + s.a$, $a_i \leftarrow a_i - s.a$
10. Else,
11.     If $a_i - u_i > 2S$,
12.         Transfer the emptiest superblock $s$
                to heap 0.
13.         $u_0 \leftarrow u_0 + s.u$, $u_i \leftarrow u_i - s.u$
14.         $a_0 \leftarrow a_0 + s.a$, $a_i \leftarrow a_i - s.a$
```

Figure 2: Pseudocode for Hoard's `free`.

3 Analysis

3.1 Notation

Before we proceed to the proof of Hoard's memory efficiency, we introduce some useful notation. Let a denote the amount of memory held in the processor heaps ($a = \sum_{i=1}^{P} a_i$). Let a^* be the total amount of memory in the processor and process heaps ($a^* = a + a_0$). When we refer to values at a certain time step, we present them as functions over time, as in $a(t)$. Let A and A^* be the maxima of a and a^* ($A(T) = \max_{t \le T} a(t)$, $A^*(T) = \max_{t \le T} a^*(t)$). Note that since we never return memory to the system, a^* never decreases, so $A^*(t) = a^*(t)$. Likewise, we define U and U^* as the maximum memory in use (since U^* is the maximum sum of u_i while U is the maximum sum of u_i for $i \ge 1$, $U \le U^*$).

In the analysis below, we first prove a lemma and a theorem that hold for any individual size class c, and then extend these results to prove a bound that holds for all $\log_B S$ size classes. We omit subscripts for size classes except in the proof of the overall bound in Theorem 2.

3.2 Memory Efficiency

In this section, we prove that $A^*(t) = O(\log(M/m)\ U^*(t))$. Robson showed that this bound holds for any uniprocessor allocator [Rob77]. By proving the equivalent bound for Hoard, we demonstrate its memory efficiency.

For the proof, we first need to show that the maximum amount of memory used in the processor heaps (heaps 1 through P) is the maximum amount of memory used in all of the heaps (heaps 0 through P), for any given size class.

Lemma 1: $A = A^*$.

Proof. As noted above, $A^* = a^*$, so we prove the equivalent assertion, $A = a^*$ by induction over the number of steps. At step 0, no memory is allocated, so $A(0) = a^*(0) = 0$. We now assume the induction hypothesis for step t and show that at step $t + 1$, $A(t + 1) = a^*(t + 1)$. We define a step as a call by one processor i to `malloc` or `free`. Neither A nor a^* are affected when a processor calls `free`, because a decreases (since we decrement a_i) while a^* remains unchanged (we subtract $s.a$ from a_i and add it to a_0).

When processor i calls `malloc`, there are three possibilities:

Case 1: There is an available superblock in heap i.

Since no memory is allocated or transferred between heaps, there is no change to either A or a^*.

Case 2: Heap 0 is empty ($a_0 = 0$).

In this case, Hoard allocates a new superblock, so $a^*(t + 1) = a^*(t) + S$. By the induction hypothesis and the definition of a^*, $A(t) = a^*(t) = a_0(t) + a(t)$. Since $a_0 = 0$, $A(t) = a^*(t) = a(t)$. The total amount held in the processor heaps increases by S with the allocation of the new superblock, so $a(t+1) = a(t) + S > A(t)$. By definition, $A(t+1) = \max\{A(t), a(t+1)\} = a(t+1)$. This, in turn, is just $a(t) + S = a^*(t+1)$, so $A(t+1) = a^*(t+1)$.

Case 3: Heap 0 is non-empty ($a_0 > 0$).

When heap 0 is non-empty, $a_0 \geq S$ (since we allocate and transfer superblocks of size S). Because no memory is allocated, $a^*(t+1) = a^*(t)$. By the definition of a^*, we have $a^*(t) = a_0(t) + a(t) \geq a(t) + S$. Transferring the superblock from heap 0 to heap i increases a by S: $a(t+1) = a(t) + S \leq a^*(t)$, which by the induction hypothesis $= A(t)$. By definition, $A(t+1) = \max\{A(t), a(t+1)\} = A(t)$, so we have $A(t+1) = a^*(t+1)$. ∎

In the rest of the analysis, we ignore "large" blocks (since these are immediately returned to the process heap, they are immediately available for re-use). For now, we also ignore the internal fragmentation that can result from rounding up to size classes (this is at most B).

We first bound Hoard's memory fragmentation for each size class:

Theorem 1: *For each size class,* $A^*(t) \leq U^*(t) + 2PS$.

Proof. Reordering the invariant as $a_i \leq u_i + 2S$ and summing over all P processor heaps gives us

$$\begin{array}{lll} A(t) & \leq & \sum_{i=1}^{P} u_i(t) + 2PS \\ & \leq & U(t) + 2PS \qquad \triangleright \textit{def. of } U(t) \\ & \leq & U^*(t) + 2PS. \qquad \triangleright U(t) \leq U^*(t) \end{array}$$

By Lemma 1 we have $A(t) = A^*(t)$, so $A^*(t) \leq U^*(t) + 2PS$. ∎

We now establish Hoard's memory efficiency:

Theorem 2: $A^*(t) = O(\log(M/m)\, U^*(t))$.

Proof. Sum Theorem 1 over the $(\log_B S)$ size classes of blocks of size S and smaller. This gives us $\sum_c A^*_c(t) \leq \sum_c U^*_c(t) + 2PS \log_B S$. Since the amount allocated never decreases, the first term can be replaced by $A^*(t)$. The maximum amount of memory in use overall is at least as large as the maximum in one of the size classes: $U^*(t) \geq \max_c U^*_c(t)$. Each of the $\log_B S$ size classes has no more than this maximum in use (otherwise, it wouldn't be the maximum), so $\sum_c U^*_c(t) \leq \log_B S \ \max_c U^*_c(t)$. To account for the internal fragmentation that can result from rounding up to powers of B, we multiply the U^* terms by B. This gives us the bound $A^*(t) \leq B \log_B S\, U^*(t) + 2PS \log_B S$. Since we are only concerned with blocks no larger than S, $M = S$ and $m = 1$, so we have $A^*(t) = O(\log(M/m)\, U^*(t))$. ∎

4 Speed and Scalability

The algorithms used by Hoard provide speed and scalability in the following ways:

Superblocks relieve contention. By allocating in superblocks of at least S bytes, we avoid many calls to the system's memory allocator (for small blocks). This relieves us of both contention (for the system's memory allocator) and many expensive system calls.

Hysteresis reduces process heap contention. Since the release threshold is the size of two empty superblocks ($2S$) and we acquire one superblock at a time, the number of local allocations and deallocations required between accesses to the process heap is likely to be proportional to S. This can be defeated by a pathological sequence of allocations and deallocations, but in practice it works well.

Most heap access is contention-free. Because each superblock is present on exactly one heap, processors never contend for allocation of blocks within a superblock. As long as a processor frees blocks that it allocated, calls to `free` only involve access to its processor heap. While a processor can free a block it allocated arbitrarily many times in a tight loop, it is significantly harder for a processor to free a block belonging to another heap. The processor must first obtain this block from another processor. This usually entails some kind of rendezvous, increasing the time interval between such operations.

Superblocks improve locality. A private-heaps allocator can produce widespread false sharing by distributing a cache line into every processor's private heap. But by allocating from superblocks, each processor tends to have exclusive use of large contiguous chunks of memory. As long as the superblock size is greater than the system's page size, page-level locality is also improved.

5 Experiments

We performed a variety of experiments on uniprocessors and multiprocessors. The platform used is a dedicated 14-processor Sun Enterprise 5000 running Solaris 7. Each processor is a 400MHz UltraSparc.

5.1 Multiprocessor Experiments

To demonstrate Hoard's speed and scalability, we compare Hoard's performance to several memory allocators:

Solaris 7 (the allocator shipped with Solaris) This is a monolithic allocator, with its heap protected by a single lock. We expect this allocator to have the lowest scalability, but we use this as a benchmark for uniprocessor performance.

Private-Heaps (a pure private-heaps allocator variant of Hoard) Despite its memory inefficiency, we include it to establish an upper-bound on scalability. Further, because it is a "brain-dead" allocator (for instance, it does no coalescing), it is extremely fast, so it is provides a reasonable upper-bound on performance.

Ptmalloc (Wolfram Gloger's subheap allocator [Glo]) This allocator has unbounded memory consumption, like the private-heaps allocator. We include it because it is the only multiprocessor allocator we know of that is in widespread use (it is the standard Linux allocator).

For each of the experiments below, we run the benchmarks three times and use the average. We use the word *speedup* for the speedup with respect to the Solaris allocator, while we use *scaleup* for the speedup of each allocator with respect to itself. Unfortunately, we are unable to measure fragmentation. We need a lock to maintain these statistics, and contention for this lock produces a dramatically different schedule of allocations and frees.

The first multithreaded benchmark we present is our own creation, called *threadtest*. This is a very simple benchmark: t threads do nothing but repeatedly allocate and deallocate