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 ← 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 {sz, S} bytes
              as superblock s
              and set the owner to heap i.
7.          a_i ← a_i + s.a.
8.    Else,
9.          Transfer the superblock s to heap i.
10.         u_0 ← u_0 − s.u
11.         u_i ← u_i + s.u
12.         a_0 ← a_0 − s.a
13.         a_i ← a_i + s.a
14. u_i ← u_i + 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^* ($A(T) = \max_{t \leq T} a(t)$, $A^*(T) = \max_{t \leq T} a^*(t)$). 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 $i \geq 1$, $U \leq U^*$).

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

K/t blocks of a given size. As seen in Figure 3, both Hoard and Private-heaps exhibit linear speedup, while the Solaris allocator exhibits severe slowdown. For 14 processors, the Hoard version runs 60% faster than the *Ptmalloc* version.

The *shbench* benchmark is available on MicroQuill's website and is shipped with the SmartHeap SMP product. Each thread repeatedly allocates and frees a number of randomly-sized blocks in random order, for a total of 50 million allocated blocks. The graphs in Figure 4 show that Hoard scales quite well, approaching linear speedup as the number of threads increases. *Ptmalloc* doesn't scale nearly as well. For 14 processors, the Hoard version runs 65% faster than the *Ptmalloc* version.

The intent of the *Larson* benchmark, due to Larson and Krishnan [mLK98], is to simulate a workload for a server. A number of threads are repeatedly spawned to allocate and free blocks in a random order. Further, a number of blocks are left to be freed by a subsequent thread. Larson and Krishnan observe this behavior (which they call "bleeding") in actual server applications, and their benchmark simulates this effect.

The *Larson* benchmark measures the throughput of the allocator. As the number of threads increases, we'd like a linear increase in the throughput. Figure 5 shows that Hoard does quite well, scaling linearly. *Ptmalloc* doesn't scale very well at all. For 14 processors, the Hoard version runs 826% faster than the *Ptmalloc* version.

In addition to benchmarks, we tested a number of applications. *Barnes-Hut* is an n-body particle solver included with Hood, a user-level multiprocessor threads library [ABBP99]. It performs a small amount of dynamic memory allocation during the tree-building phase. With 14 processors, all of the scalable allocators provide about a 10% performance improvement,

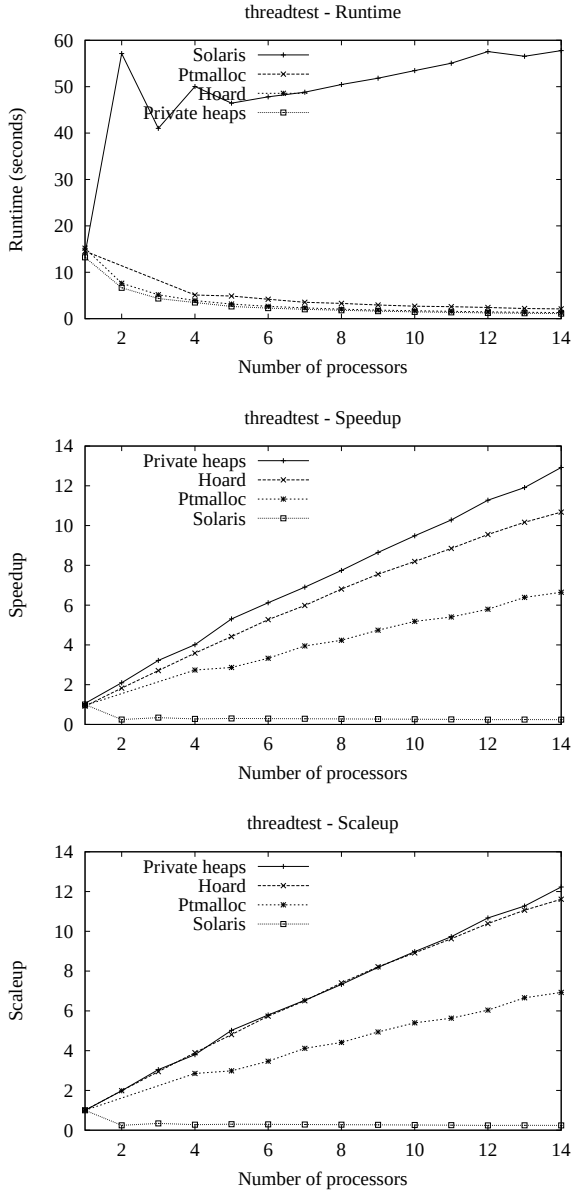
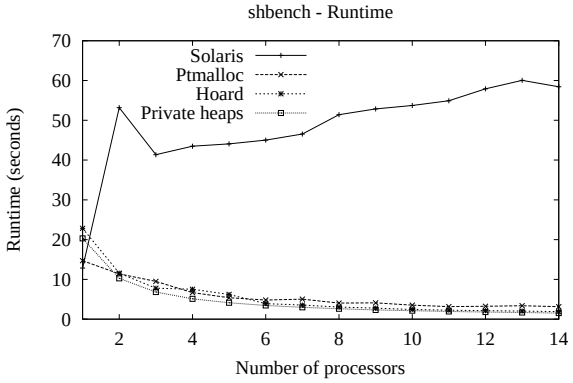
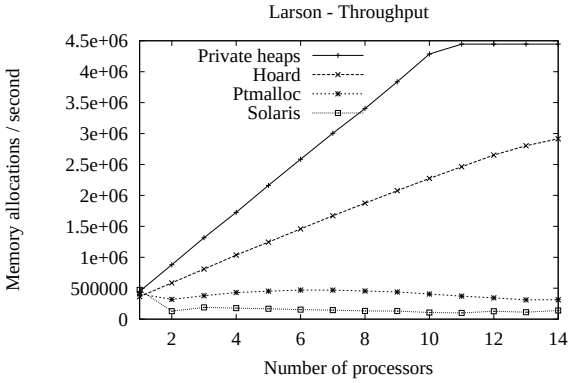
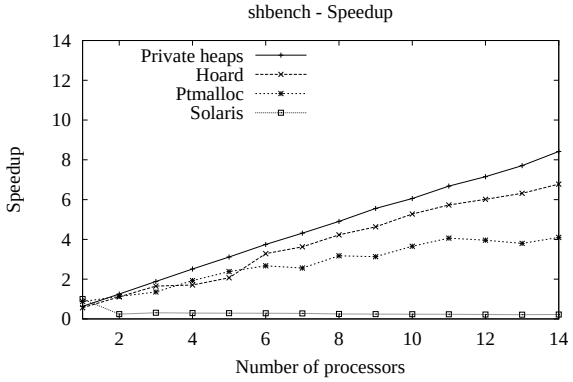
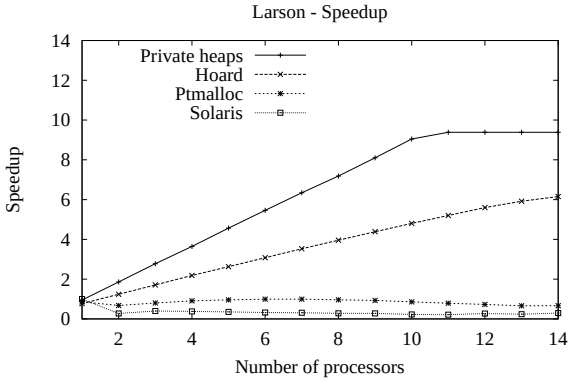


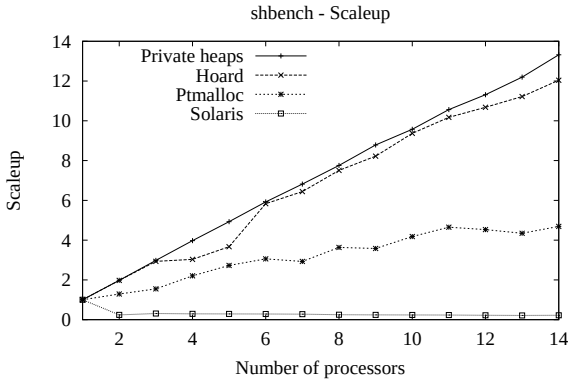
Figure 3: Runtime, speedup and scaleup using the Threadtest benchmark.











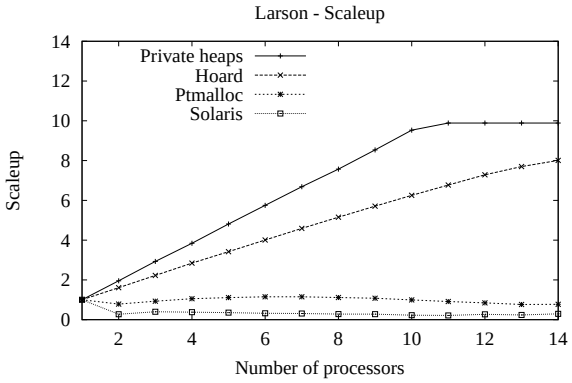


Figure 4: Runtime, speedup and scaleup using the SmartHeap benchmark.

Figure 5: Allocation throughput, speedup and scaleup using the Larson benchmark.

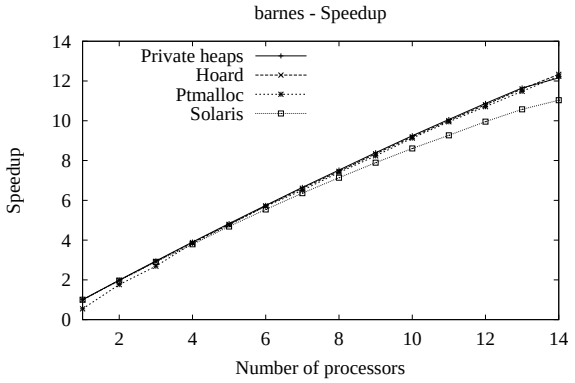


Figure 6: Barnes-Hut speedup.

increasing the speedup of the application from 11 to just above 12 (see Figure 6).

mm is a multithreaded recursive matrix-multiply code that uses Hood. It performs a moderate amount of memory allocation relative to the amount of computation, allocating temporary matrices to hold intermediate results. Figure 7 shows that both Hoard and PrivateHeaps improve speedup. For 14 processors, Hoard increases the speedup by about 20% over the Solaris allocator. (We were unable to include data for runs with *ptmalloc*, which did not work with *mm*.)

5.2 Uniprocessor Experiments

To show that Hoard consumes a reasonable amount of memory even when used as a uniprocessor allocator, we linked Hoard with the following applications described by Wilson and Johnstone [JW98]: *espresso*, an optimizer for programmable logic arrays; *Ghostscript*, a PostScript interpreter; *Hyper*, a hypercube network communication simulator; *LRUsim*, a locality analyzer, and *p2c*, a Pascal-to-C translator. These programs were chosen specifically because they are allocation-intensive but have widely varying memory usage patterns.

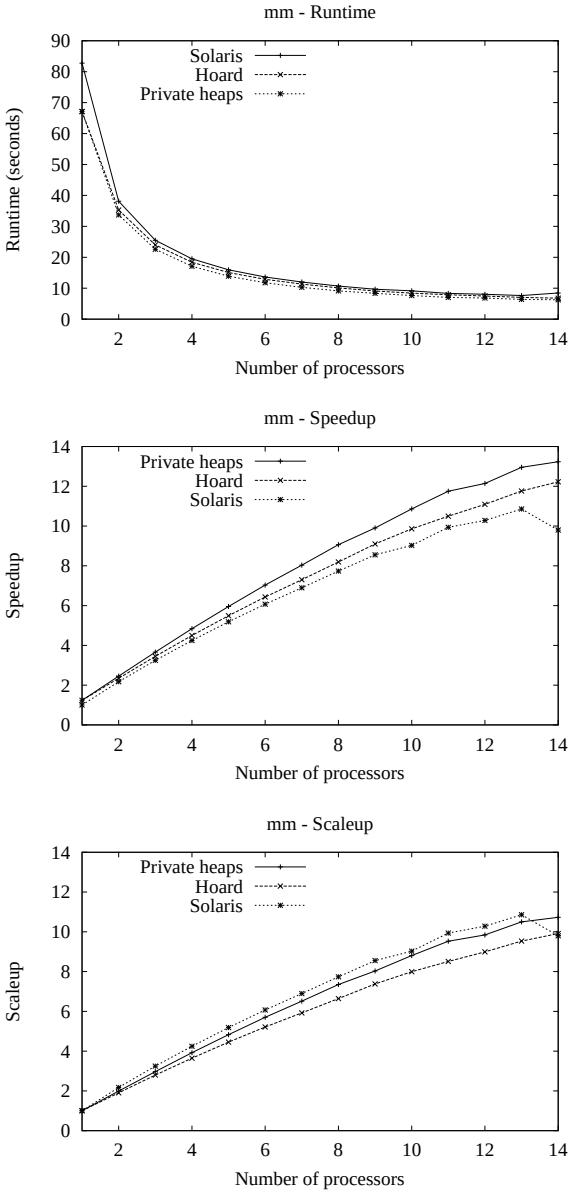


Figure 7: Runtime, speedup and scaleup using *mm* (a multithreaded matrix-multiply code).

program	Hoard frag.	[Lea] frag.	footprint (Kbytes)
espresso	4.8359	1.0026	270
Ghostscript	1.2154	1.034	1,136
Hyper	1.1056	1.0016	1,853
LRUsim	1.0513	1.0026	1,530
p2c	1.0060	1.0178	416

Table 1: Fragmentation using Hoard with uniprocessor applications along with the footprints (maximum memory used) for each application. Included for comparison is fragmentation using Doug Lea's allocator.

In Table 1, we report *fragmentation* (the maximum amount of memory allocated divided by the program's *footprint*, the maximum amount of memory in use at any time) for both Hoard and Doug Lea's allocator [JW98, Lea]. Hoard's increase in memory consumption over Doug Lea's allocator is generally between 0% and 18%, except for *espresso*, which allocates a small amount of memory from a large number of different size classes. Because our superblocks are relatively large (32K) compared with *espresso*'s footprint (270K), the overhead of using superblocks is significant (although it increases the total memory consumption to just one megabyte).

6 Previous Work

We are aware of a number of multiprocessor allocators, including those described in the literature and one that is publicly available on the Web. Only one appears to be in widespread use – Gloger's *ptmalloc*, used in Linux. Most other operating systems, including Solaris 7, FreeBSD [Kam98], and Windows NT 4[1][Kri99] use a monolithic allocator.

It is convenient to divide these allocators into two classes: *single* and *multiple* heap allocators. Single-heap allocators use a single concurrent heap [Sto82, EO88, Joh91, JD92, Iye92]. Multiple-heap allocators statically or dynamically assign a heap to each processor [Sah91, BL94, Glo, mLK98, Tzv99].

6.1 Single-Heap Allocators

The single-heap allocators are generally adaptations of well-known uniprocessor allocator algorithms, like first-fit, best-fit, and fast-fit, replacing the single heap lock by fine-grained locks or atomic operations.

In the absence of contention, concurrent single-heap allocators are generally much slower than monolithic allocators. For a good uniprocessor allocator like Doug Lea's [Lea], an uncontended lock acquisition and release takes as long as any allocation or deallocation operation. Acquiring and releasing just one additional uncontended lock would make the allocator run 50% slower. Atomic operations are also prohibitively expensive. If we replaced the lock with just five 64-bit `compare-and-swap` operations (on a Pentium III), the allocator would run 28% slower.

Because the overhead of locking dominates the cost of allocators, any allocation scheme that requires more than one lock is too slow to be practical except in the presence of high contention. This includes the allocation algorithms described by Johnson and Iyengar [Joh91, JD92, Iye92] which use concurrent trees to represent the heap and so typically require $O(\log S)$ lock operations, where S is the number of size classes used by the program. Stone's allocator uses *fetch-and-*ϕ to simulate read and write locks in a first-fit allocator [Sto82]; this too is prohibitively expensive. Ellis and Olson

[1]The allocator in Service Pack 4 and beyond (to be included in Windows 2000) uses 64-bit atomic operations instead of locks for the small block freelists, but still uses one central heap.

describe two concurrent single-heap algorithms [EO88]. The first uses locks for every block in the freelist (and their experiments show it to be exceedingly expensive), while the second uses an optimistic protocol to avoid locking, but they show that this algorithm does not scale. For small blocks, Iyengar uses *freelist locks*: for each size class, there is a separate freelist, protected by a lock [Iye92]. This reduces contention as long as the requests are for blocks of varied sizes, but yields no improvement for allocations of same-sized blocks. Wilson and Johnstone observe that "for most programs, the vast majority of objects allocated are of only a few sizes" [JW98], so freelist locks are of limited value.

6.2 Multiple Heaps

Because single-heap allocators have so far proven to be too inefficient to be practical, most recent work on multiprocessor allocators has focused on multiple-heap allocators. Unfortunately, these typically have serious memory fragmentation problems (like the unbounded memory consumption in pure private-heaps allocators).

Gloger describes an algorithm that implements what we call a *subheap* allocator [Glo]. There are a number of distinct heaps called subheaps, each protected by a lock. The allocator uses the first unlocked subheap for both allocation and deallocation. If there is no unlocked subheap, a new subheap is created. When a subheap is initialized or emptied, it is refilled with a large block from the global heap. Using subheaps avoids contention on locks, but memory consumption is unbounded. If there are two threads running, a producer thread can always end up allocating from the first subheap and the consumer can always free to the second (if the producer thread holds the lock on the first subheap while the consumer attempts to free its block), leading to the same unbounded memory consumption described above for pure private-heaps.

Another way to avoid unbounded memory consumption for a private-heaps allocator is to return freed blocks to the heap they were allocated from. This approach is used by Larson and Krishnan [mLK98]. While this fixes unbounded memory consumption, it leads to a P-fold increase in memory consumption, as in the following round-robin style producer-consumer example: each processor i allocates K blocks and processor $i + 1$ frees them (processor $P - 1$'s blocks are freed by processor 0). This allocates PK blocks (K on all P freelists); a centralized allocator would have allocated just K blocks.

One fix for the unbounded memory consumption problem, implemented by Tzvetkov [Tzv99], requires that each processor keep no more than $2T$ bytes of free memory on any of its per-size class freelists (Vee and Hsu do the same in their one size-class allocator [VH99]). A problem with Tzvetkov's allocator is that it follows the same release policy for "large" blocks (size $\geq T$) as for "small" blocks (size $< T$): one large block is always left on a processor's heap when two are freed. This yields a P-fold increase in worst-case memory consumption. Modifying Tzvetkov's allocator so that it immediately returns large free blocks to the global heap makes it memory efficient.

To limit false sharing, Tzvetkov pads blocks with 64 extra bytes (a comment in the code notes that this doubles the performance of certain programs). For 8-byte blocks, this amounts to 900% internal fragmentation, which is too high to be practical for many applications.

7 Conclusion

We show that Hoard, a multiprocessor memory allocator, is provably memory-efficient. 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 matches the lower bound for worst-case fragmentation that holds for uniprocessor allocators [Rob77]. On uniprocesors, we show that Hoard's fragmentation is reasonable; for 4 out of 5 benchmarks, Hoard consumes no more than 18% more memory than a well-known uniprocessor allocator [Lea]. We also show empirically that Hoard is fast and scalable. Hoard scales linearly with the number of processors for every benchmark and application we tested. For three benchmarks, Hoard is between 60% and 826% faster than the standard Linux multiprocessor allocator when running on 14 processors.

8 Acknowledgements

Many thanks to Paul Wilson, Greg Plaxton, Yannis Smaragdakis, Rich Cardone, Scott Kaplan, Phoebe Weidmann and Brendon Cahoon for valuable discussions during the course of this work and input during the writing of this paper.

References

[ABBP99] Umut Acar, Emery Berger, Robert Blumofe, and Dionysios Papadopoulos. Hood: A threads library for multiprogrammed multiprocessors. http://www.cs.utexas.edu/users/hood, September 1999.

[BL94] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. In *Proceedings of the 35th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 356–368, Santa Fe, New Mexico, November 1994.

[EO88] Carla Schlatter Ellis and Thomas J. Olson. Algorithms for parallel memory allocation. *International Journal of Parallel Programming*, 17(4):303–345, 1988.

[Glo] Wolfram Gloger. Dynamic memory allocator implementations in linux system libraries. http://www.dent.med.uni-muenchen.de/~ wmglo/malloc-slides.html.

[Iye92] Arun K. Iyengar. *Dynamic Storage Allocation on a Multiprocessor*. PhD thesis, MIT, 1992. MIT Laboratory for Computer Science Technical Report MIT/LCS/TR–560.

[JD92] Theodore Johnson and Tim Davis. Space efficient parallel buddy memory management. Technical Report TR92-008, University of Florida, Department of CIS, 1992.

[Joh91] T. Johnson. A concurrent fast-fits memory manager. Technical Report TR91-009, University of Florida, Department of CIS, 1991.

[JW98] Mark S. Johnstone and Paul R. Wilson. The memory fragmentation problem: Solved? In *ISMM*, Vancouver, B.C., Canada, 1998.

[Kam98] Poul-Henning Kamp. `Malloc(3)` revisited. USENIX Freenix Track, 1998.

[Kri99] Murali R. Krishnan. Heap: Pleasures and pains. Microsoft Developer Newsletter, February 1999.

[Lea] Doug Lea. A memory allocator. http://g.oswego.edu/dl/html/malloc.html.

[mLK98] Per Åke Larson and Murali Krishnan. Memory allocation for long-running server applications. In *ISMM*, Vancouver, B.C., Canada, 1998.

[Rob77] J. M. Robson. Worst case fragmentation of first fit and best fit storage allocation strategies. *ACM Computer Journal*, 20(3):242–244, August 1977.

[Sah91] Anuragh Sah. Parallel language support on shared memory multiprocessors. Master's thesis, University of California, Berkeley - Department Of Computer Science, May 1991.

[Sto82] H. Stone. Parallel memory allocation using the FETCH-AND-ADD instruction. Technical Report RC 9674, IBM T. J. Watson Research Center, November 1982.

[Tzv99] Svetoslav Tzvetkov. Private communication. January 1999.

[VH99] Voon-Yee Vee and Wen-Jing Hsu. A scalable and efficient storage allocator on shared-memory multiprocessors. In *International Symposium on Parallel Architectures, Algorithms, and Networks (I-SPAN'99)*, pages 230–235, Fremantle, Western Australia, June 1999.