Hoard: A Scalable Memory Allocator for Multithreaded Applications

Emery D. Berger* Kathryn S. McKinley† Robert D. Blumofe* Paul R. Wilson*

*Department of Computer Sciences
The University of Texas at Austin
Austin, Texas 78712
{emery, rdb, wilson}@cs.utexas.edu

†Department of Computer Science
University of Massachusetts
Amherst, Massachusetts 01003
mckinley@cs.umass.edu

ABSTRACT

Parallel, multithreaded C and C++ programs such as web servers, database managers, news servers, and scientific applications are becoming increasingly prevalent. For these applications, the memory allocator is often a bottleneck that severely limits program performance and scalability on multiprocessor systems. Previous allocators suffer from problems that include poor performance and scalability, and heap organizations that introduce false sharing. Worse, many allocators exhibit a dramatic increase in memory consumption when confronted with a producer-consumer pattern of object allocation and freeing. This increase in memory consumption can range from a factor of P (the number of processors) to unbounded memory consumption.

This paper introduces Hoard, a fast, highly scalable allocator that largely avoids false sharing and is memory efficient. Hoard is the first allocator to simultaneously solve the above problems. Hoard combines one global heap and per-processor heaps with a novel discipline that provably bounds memory consumption and has very low synchronization costs in the common case. Our results on eleven programs demonstrate that Hoard yields low average fragmentation and improves overall program performance over the standard Solaris allocator by up to a factor of 60 on 14 processors, and up to a factor of 18 over the next best allocator we tested.

1. Introduction

Parallel, multithreaded programs are becoming increasingly prevalent. These applications include web servers [35], database managers [27], news servers [3], as well as more traditional parallel applications such as scientific applications [7]. For these applications, high performance is critical. They are generally written in C or C++ to run efficiently on modern shared-memory multiprocessor servers. Many of these applications make intensive use of dynamic memory allocation. Unfortunately, the memory allocator is often a bottleneck that severely limits program scalability on multiprocessor systems [21]. Existing serial memory allocators do not scale well for multithreaded applications, and existing concurrent allocators do not provide one or more of the following features, all of which are needed in order to attain scalable and memory-efficient allocator performance:

Speed. A memory allocator should perform memory operations (i.e., malloc and free) about as fast as a state-of-the-art serial memory allocator. This feature guarantees good allocator performance even when a multithreaded program executes on a single processor.

Scalability. As the number of processors in the system grows, the performance of the allocator must scale linearly with the number of processors to ensure scalable application performance.

False sharing avoidance. The allocator should not introduce false sharing of cache lines in which threads on distinct processors inadvertently share data on the same cache line.

Low fragmentation. We define *fragmentation* as the maximum amount of memory allocated from the operating system divided by the maximum amount of memory required by the application. Excessive fragmentation can degrade performance by causing poor data locality, leading to paging.

Certain classes of memory allocators (described in Section 6) exhibit a special kind of fragmentation that we call *blowup*. Intuitively, blowup is the increase in memory consumption caused when a concurrent allocator reclaims memory freed by the program but fails to use it to satisfy future memory requests. We define blowup as the maximum amount of memory allocated by a given allocator divided by the maximum amount of memory allocated by an ideal uniprocessor allocator. As we show in Section 2.2, the common producer-consumer programming idiom can cause blowup. In many allocators, blowup ranges from a factor of P (the number of processors) to unbounded memory consumption (the longer the program runs, the more memory it consumes). Such a pathological increase in memory consumption can be catastrophic, resulting in premature application termination due to exhaustion of swap space.

The contribution of this paper is to introduce the Hoard allocator and show that it enables parallel multithreaded programs to achieve scalable performance on shared-memory multiprocessors. Hoard achieves this result by simultaneously solving all of the above problems. In particular, Hoard solves the blowup and false sharing problems, which, as far as we know, have never been addressed in the

This work is 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. Kathryn McKinley was supported by DARPA Grant 5-21425, NSF Grant EIA-9726401, and NSF CAREER Award CCR-9624209. In addition, Emery Berger was supported by a Novell Corporation Fellowship. Multiprocessor computing facilities were provided through a generous donation by Sun Microsystems.

ASPLOS 2000 **Cambridge, MA** **Nov. 12-15 , 2000**

literature. As we demonstrate, Hoard also achieves nearly zero synchronization costs in practice.

Hoard maintains per-processor heaps and one global heap. When a per-processor heap's usage drops below a certain fraction, Hoard transfers a large fixed-size chunk of its memory from the per-processor heap to the global heap, where it is then available for reuse by another processor. We show that this algorithm bounds blowup and synchronization costs to a constant factor. This algorithm avoids false sharing by ensuring that the same processor almost always reuses (i.e., repeatedly mallocs) from a given cache line. Results on eleven programs demonstrate that Hoard scales linearly as the number of processors grows and that its fragmentation costs are low. On 14 processors, Hoard improves performance over the standard Solaris allocator by up to a factor of 60 and a factor of 18 over the next best allocator we tested. These features have led to its incorporation in a number of high-performance commercial applications, including the Twister, Typhoon, Breeze and Cyclone chat and USENET servers [3] and BEMSolver, a high-performance scientific code [7].

The rest of this paper is organized as follows. In Section 2, we explain in detail the issues of blowup and allocator-induced false sharing. In Section 3, we motivate and describe in detail the algorithms used by Hoard to simultaneously solve these problems. We sketch proofs of the bounds on blowup and contention in Section 4. We demonstrate Hoard's speed, scalability, false sharing avoidance, and low fragmentation empirically in Section 5, including comparisons with serial and concurrent memory allocators. We also show that Hoard is robust with respect to changes to its key parameter. We classify previous work into a taxonomy of memory allocators in Section 6, focusing on speed, scalability, false sharing and fragmentation problems described above. Finally, we discuss future directions for this research in Section 7, and conclude in Section 8.

2. Motivation

In this section, we focus special attention on the issues of allocator-induced false sharing of heap objects and blowup to motivate our work. These issues must be addressed to achieve efficient memory allocation for scalable multithreaded applications but have been neglected in the memory allocation literature.

2.1 Allocator-Induced False Sharing of Heap Objects

False sharing occurs when multiple processors share words in the same cache line without actually sharing data and is a notorious cause of poor performance in parallel applications [20, 15, 36]. Allocators can cause false sharing of heap objects by dividing cache lines into a number of small objects that distinct processors then write. A program may introduce false sharing by allocating a number of objects within one cache line and passing an object to a different thread. It is thus impossible to completely avoid false sharing of heap objects unless the allocator pads out every memory request to the size of a cache line. However, no allocator we know of pads memory requests to the size of a cache line, and with good reason; padding could cause a dramatic increase in memory consumption (for instance, objects would be padded to a multiple of 64 bytes on a SPARC) and could significantly degrade spatial locality and cache utilization.

Unfortunately, an allocator can *actively induce* false sharing even on objects that the program does not pass to different threads. Active false sharing is due to malloc satisfying memory requests by different threads from the same cache line. For instance, single-heap allocators can give many threads parts of the same cache line. The allocator may divide a cache line into 8-byte chunks. If multiple threads request 8-byte objects, the allocator may give each thread one 8-byte object in turn. This splitting of cache lines can lead to false sharing.

Allocators may also *passively induce* false sharing. Passive false sharing occurs when free allows a future malloc to produce false sharing. If a *program* introduces false sharing by spreading the pieces of a cache line across processors, the allocator may then passively induce false sharing after a free by letting each processor reuse pieces it freed, which can then lead to false sharing.

2.2 Blowup

Many previous allocators suffer from blowup. As we show in Section 3.1, Hoard keeps blowup to a constant factor but many existing concurrent allocators have *unbounded* blowup (the Cilk and STL allocators [6, 30]) (memory consumption grows without bound while the memory required is fixed) or memory consumption can grow linearly with P, the number of processors (Ptmalloc and LKmalloc [9, 22]). It is important to note that these worst cases are not just theoretical. Threads in a producer-consumer relationship, a common programming idiom, may induce this blowup. To the best of our knowledge, papers in the literature do not address this problem. For example, consider a program in which a producer thread repeatedly allocates a block of memory and gives it to a consumer thread which frees it. If the memory freed by the consumer is unavailable to the producer, the program consumes more and more memory as it runs.

This unbounded memory consumption is plainly unacceptable, but a P-fold increase in memory consumption is also cause for concern. The scheduling of multithreaded programs can cause them to require *much* more memory when run on multiple processors than when run on one processor [6, 28]. Consider a program with P threads. Each thread calls x=malloc(s); free(x). If these threads are serialized, the total memory required is s. However, if they execute on P processors, each call to malloc may run in parallel, increasing the memory requirement to $P * s$. If the allocator multiplies this consumption by another factor of P, then memory consumption increases to $P^2 * s$.

3. The Hoard Memory Allocator

This section describes Hoard in detail. Hoard can be viewed as an allocator that generally avoids false sharing and that trades increased (but bounded) memory consumption for reduced synchronization costs.

Hoard augments per-processor heaps with a *global heap* that every thread may access (similar to Vee and Hsu [37]). Each thread can access only its heap and the global heap. We designate heap 0 as the global heap and heaps 1 through P as the per-processor heaps. In the implementation we actually use $2P$ heaps (without altering our analytical results) in order to decrease the probability that concurrently-executing threads use the same heap; we use a simple hash function to map thread id's to per-processor heaps that can result in collisions. We need such a mapping function because in general there is not a one-to-one correspondence between threads and processors, and threads can be reassigned to other processors. On Solaris, however, we are able to avoid collisions of heap assignments to threads by hashing on the light-weight process (LWP) id. The number of LWP's is usually set to the number of processors [24, 33], so each heap is generally used by no more than one LWP.

Hoard maintains *usage statistics* for each heap. These statistics are u_i, the amount of memory in use ("live") in heap i, and a_i, the amount of memory allocated by Hoard from the operating system held in heap i.

Hoard allocates memory from the system in chunks we call *superblocks*. Each superblock is an array of some number of blocks (objects) and contains a free list of its available blocks maintained in LIFO order to improve locality. All superblocks are the same

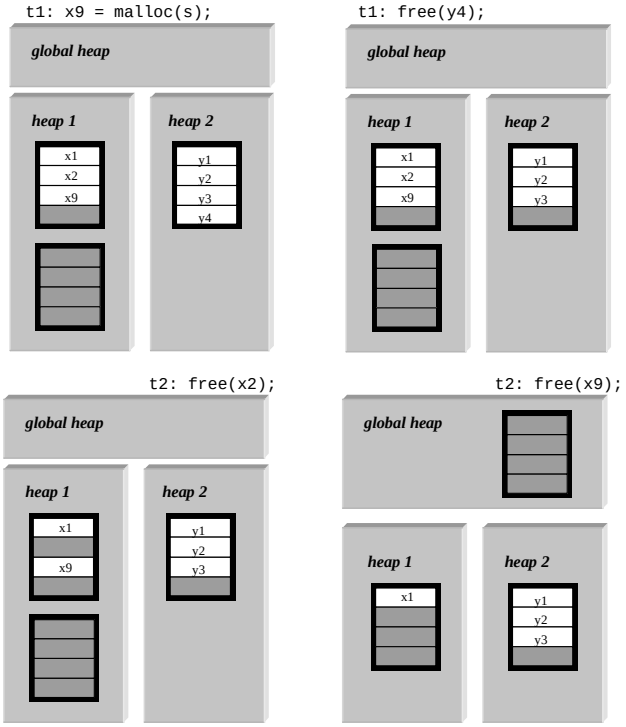


Figure 1: Allocation and freeing in Hoard. See Section 3.2 for details.

size (S), a multiple of the system page size. Objects larger than half the size of a superblock are managed directly using the virtual memory system (i.e., they are allocated via `mmap` and freed using `munmap`). All of the blocks in a superblock are in the same size class. By using size classes that are a power of b apart (where b is greater than 1) and rounding the requested size up to the nearest size class, we bound worst-case *internal* fragmentation within a block to a factor of b. In order to reduce *external* fragmentation, we *recycle* completely empty superblocks for re-use by any size class. For clarity of exposition, we assume a single size class in the discussion below.

3.1 Bounding Blowup

Each heap "owns" a number of superblocks. When there is no memory available in any superblock on a thread's heap, Hoard obtains a superblock from the global heap if one is available. If the global heap is also empty, Hoard creates a new superblock by requesting virtual memory from the operating system and adds it to the thread's heap. Hoard does not currently return empty superblocks to the operating system. It instead makes these superblocks available for reuse.

Hoard moves superblocks from a per-processor heap to the global heap when the per-processor heap crosses the *emptiness threshold*: more than f, the *empty fraction*, of its blocks are not in use ($u_i < (1-f)a_i$), and there are more than some number K of superblocks' worth of free memory on the heap ($u_i < a_i - K * S$). As long as a heap is not more than f empty, and has K or fewer superblocks, Hoard will not move superblocks from a per-processor heap to the global heap. Whenever a per-processor heap does cross the emptiness threshold, Hoard transfers one of its superblocks that is at least f empty to the global heap. Always removing such a superblock whenever we cross the emptiness threshold maintains the following invariant on the per-processor heaps: $(u_i \geq a_i - K * S) \wedge (u_i \geq (1-f)a_i)$. When we remove a superblock, we reduce u_i by at most $(1-f)S$ but reduce a_i by S, thus restoring the invariant. Maintaining this invariant bounds blowup to a constant factor, as we show in Section 4.

Hoard finds f-empty superblocks in constant time by dividing superblocks into a number of bins that we call "fullness groups". Each bin contains a doubly-linked list of superblocks that are in a given fullness range (e.g., all superblocks that are between $3/4$ and completely empty are in the same bin). Hoard moves superblocks from one group to another when appropriate, and always allocates from nearly-full superblocks. To improve locality, we order the superblocks within a fullness group using a move-to-front heuristic. Whenever we free a block in a superblock, we move the superblock to the front of its fullness group. If we then need to allocate a block, we will be likely to reuse a superblock that is already in memory; because we maintain the free blocks in LIFO order, we are also likely to reuse a block that is already in cache.

3.2 Example

Figure 1 illustrates, in simplified form, how Hoard manages superblocks. For simplicity, we assume there are two threads and heaps (thread i maps to heap i). In this example (which reads from top left to top right, then bottom left to bottom right), the empty fraction f is $1/4$ and K is 0. Thread 1 executes code written on the left-hand side of each diagram (prefixed by "t1:") and thread 2 executes code on the right-hand side (prefixed by "t2:"). Initially, the global heap is empty, heap 1 has two superblocks (one partially full, one empty), and heap 2 has a completely-full superblock.

The top left diagram shows the heaps after thread 1 allocates `x9` from heap 1. Hoard selects the fullest superblock in heap 1 for allocation. Next, in the top right diagram, thread 1 frees `y4`, which is in a superblock that heap 2 owns. Because heap 2 is still more than $1/4$ full, Hoard does not remove a superblock from it. In the bottom left diagram, thread 2 frees `x2`, which is in a superblock owned by heap 1. This free does not cause heap 1 to cross the emptiness threshold, but the next free (of `x9`) does. Hoard then moves the completely-free superblock from heap 1 to the global heap.

3.3 Avoiding False Sharing

Hoard uses the combination of superblocks and multiple-heaps described above to avoid most active and passive false sharing. Only one thread may allocate from a given superblock since a superblock is owned by exactly one heap at any time. When multiple threads make simultaneous requests for memory, the requests will always be satisfied from different superblocks, avoiding actively induced false sharing. When a program deallocates a block of memory, Hoard returns the block to its superblock. This coalescing prevents multiple threads from reusing pieces of cache lines that were passed to these threads by a user program, avoiding passively-induced false sharing.

While this strategy can greatly reduce allocator-induced false sharing, it does not completely avoid it. Because Hoard may move superblocks from one heap to another, it is possible for two heaps to share cache lines. Fortunately, superblock transfer is a relatively infrequent event (occurring only when a per-processor heap has dropped below the emptiness threshold). Further, we have observed that in practice, superblocks released to the global heap are often completely empty, eliminating the possibility of false sharing. Released superblocks are guaranteed to be at least f empty, so the opportunity for false sharing of lines in each superblock is reduced.

Figure 1 also shows how Hoard generally avoids false sharing. Notice that when thread 1 frees `y4`, Hoard returns this memory to

malloc (sz)
1. If sz $> S/2$, allocate the superblock from the OS
   and *return* it.
2. $i \leftarrow$ hash(the current thread).
3. Lock heap i.
4. Scan heap i's list of superblocks from most full to least
   (for the size class corresponding to sz).
5. If there is no superblock with free space,
6. Check heap 0 (the global heap) for a superblock.
7. If there is none,
8. Allocate S bytes as superblock s
   and set the owner to heap i.
9. Else,
10. Transfer the superblock s to heap i.
11. $u_0 \leftarrow u_0 - s.u$
12. $u_i \leftarrow u_i + s.u$
13. $a_0 \leftarrow a_0 - S$
14. $a_i \leftarrow a_i + S$
15. $u_i \leftarrow u_i +$ sz.
16. $s.u \leftarrow s.u +$ sz.
17. Unlock heap i.
18. Return a block from the superblock.

free (ptr)
1. If the block is "large",
2. Free the superblock to the operating system and *return*.
3. Find the superblock s this block comes from and lock it.
4. Lock heap i, the superblock's owner.
5. Deallocate the block from the superblock.
6. $u_i \leftarrow u_i -$ block size.
7. $s.u \leftarrow s.u -$ 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 heap 0 (the global heap).
11. $u_0 \leftarrow u_0 + s1.u$, $u_i \leftarrow u_i - s1.u$
12. $a_0 \leftarrow a_0 + S$, $a_i \leftarrow a_i - S$
13. Unlock heap i and the superblock.

Figure 2: Pseudo-code for Hoard's malloc and free.

y4's superblock and not to thread 1's heap. Since Hoard always uses heap i to satisfy memory allocation requests from thread i, only thread 2 can reuse that memory. Hoard thus avoids both active and passive false sharing in these superblocks.

3.4 Algorithms

In this section, we describe Hoard's memory allocation and deallocation algorithms in more detail. We present the pseudo-code for these algorithms in Figure 2. For clarity of exposition, we omit discussion of the management of fullness groups and superblock recycling.

Allocation

Hoard directly allocates "large" objects (size $> S/2$) via the virtual memory system. When a thread on processor i calls malloc for small objects, Hoard locks heap i and gets a block of a superblock with free space, if there is one on that heap (line 4). If there is not, Hoard checks the global heap (heap 0) for a superblock. If there is one, Hoard transfers it to heap i, adding the number of bytes in use in the superblock $s.u$ to u_i, and the total number of bytes in the superblock S to a_i (lines 10–14). If there are no superblocks in either heap i or heap 0, Hoard allocates a new superblock and inserts it into heap i (line 8). Hoard then chooses a single block from a superblock with free space, marks it as allocated, and returns a pointer to that block.

Deallocation

Each superblock has an "owner" (the processor whose heap it's in). When a processor frees a block, Hoard finds its superblock (through a pointer in the block's header). (If this block is "large", Hoard immediately frees the superblock to the operating system.) It first locks the superblock and then locks the owner's heap. Hoard then returns the block to the superblock and decrements u_i. If the heap is too empty ($u_i < a_i - K * S$ or $u_i < (1-f)a_i$), Hoard transfers a superblock that is at least f empty to the global heap (lines 10-12). Finally, Hoard unlocks heap i and the superblock.

4. Analytical Results

In this section, we sketch the proofs of bounds on blowup and synchronization. We first define some useful notation. We number the heaps from 0 to P: 0 is the global heap, and 1 through P are the per-processor heaps. We adopt the following convention: capital letters denote maxima and lower-case letters denote current values. Let $A(t)$ and $U(t)$ denote the *maximum* amount of memory allocated and in use by the program ("live memory") after memory operation t. Let $a(t)$ and $u(t)$ denote the *current* amount of memory allocated and in use by the program after memory operation t. We add a subscript for a particular heap (e.g., $u_i(t)$) and add a caret (e.g., $\hat{a}(t)$) to denote the sum for all heaps *except* the global heap.

4.1 Bounds on Blowup

We now formally define the blowup for an allocator as its worst-case memory consumption divided by the ideal worst-case memory consumption for a serial memory allocator (a constant factor times its maximum memory required [29]):

DEFINITION 1. *blowup* $= O(A(t)/U(t))$.

We first prove the following theorem that bounds Hoard's worst-case memory consumption: $A(t) = O(U(t) + P)$. We can show that the maximum amount of memory in the global and the per-processor heaps ($A(t)$) is the same as the maximum allocated into the per-processor heaps ($\hat{A}(t)$). We make use of this lemma, whose proof is straightforward but somewhat lengthy (the proof may be found in our technical report [4]).

LEMMA 1. $A(t) = \hat{A}(t)$.

Intuitively, this lemma holds because these quantities are maxima; any memory in the global heap was originally allocated into a per-processor heap. Now we prove the bounded memory consumption theorem:

THEOREM 1. $A(t) = O(U(t) + P)$.

PROOF. We restate the invariant from Section 3.1 that we maintain over all the per-processor heaps: $(a_i(t) - K * S \leq u_i(t)) \wedge ((1-f)a_i(t) \leq u_i(t))$.

The first inequality is sufficient to prove the theorem. Summing over all P per-processor heaps gives us

$$\begin{aligned}
\hat{A}(t) &\leq \textstyle\sum_{i=1}^{P} u_i(t) + P * K * S && \triangleright \textit{ def. of } \hat{A}(t) \\
&\leq \hat{U}(t) + P * K * S && \triangleright \textit{ def. of } \hat{U}(t) \\
&\leq U(t) + P * K * S. && \triangleright \hat{U}(t) \leq U(t)
\end{aligned}$$

Since by the above lemma $A(t) = \hat{A}(t)$, we have $A(t) = O(U(t) + P)$. $\square$

Because the number of size classes is constant, this theorem holds over all size classes. By the definition of blowup above, and assuming that $P << U(t)$, Hoard's blowup is $O((U(t) + P)/U(t)) = O(1)$. This result shows that Hoard's worst case memory consumption is at worst a constant factor overhead that does not grow with the amount of memory required by the program.

Our discipline for using the empty fraction (f) enables this proof, so it is clearly a key parameter for Hoard. For reasons we describe and validate with experimental results in Section 5.5, Hoard's performance is robust with respect to the choice of f.

4.2 Bounds on Synchronization

In this section, we analyze Hoard's worst-case and discuss expected synchronization costs. Synchronization costs come in two flavors: contention for a per-processor heap and acquisition of the global heap lock. We argue that the first form of contention is not a scalability concern, and that the second form is rare. Further, for common program behavior, the synchronization costs are low over most of the program's lifetime.

Per-processor Heap Contention

While the worst-case contention for Hoard arises when one thread allocates memory from the heap and a number of other threads free it (thus all contending for the same heap lock), this case is not particularly interesting. If an application allocates memory in such a manner and the amount of work between allocations is so low that heap contention is an issue, then the application itself is fundamentally unscalable. Even if heap access were to be completely independent, the application itself could only achieve a two-fold speedup, no matter how many processors are available.

Since we are concerned with providing a scalable allocator for scalable applications, we can bound Hoard's worst case for such applications, which occurs when pairs of threads exhibit the producer-consumer behavior described above. Each malloc and each free will be serialized. Modulo context-switch costs, this pattern results in at most a two-fold slowdown. This slowdown is not desirable but it is scalable as it does not grow with the number of processors (as it does for allocators with one heap protected by a single lock).

It is difficult to establish an expected case for per-processor heap contention. Since most multithreaded applications use dynamically-allocated memory for the exclusive use of the allocating thread and only a small fraction of allocated memory is freed by another thread [22], we expect per-processor heap contention to be quite low.

Global Heap Contention

Global heap contention arises when superblocks are first created, when superblocks are transferred to and from the global heap, and when blocks are freed from superblocks held by the global heap. We simply count the number of times the global heap's lock is acquired by each thread, which is an upper-bound on contention. We analyze two cases: a growing phase and a shrinking phase. We show that worst-case synchronization for the growing phases is inversely proportional to the superblock size and the empty fraction but we show that the worst-case for the shrinking phase is expensive but only for a pathological case that is unlikely to occur in practice. Empirical evidence from Section 5 suggests that for most programs, Hoard will incur low synchronization costs for most of the program's execution.

Two key parameters control the worst-case global heap contention while a per-processor heap is growing: f, the empty fraction, and S, the size of a superblock. When a per-processor heap is growing, a thread can acquire the global heap lock at most $k/(f * S/s)$ times for k memory operations, where f is the empty fraction, S is the superblock size, and s is the object size. Whenever the per-processor heap is empty, the thread will lock the global heap and obtain a superblock with at least $f * S/s$ free blocks. If the thread then calls malloc k times, it will exhaust its heap and acquire the global heap lock at most $k/(f * S/s)$ times.

When a per-processor heap is shrinking, a thread will first acquire the global heap lock when the release threshold is crossed. The release threshold could then be crossed on every single call to free if every superblock is exactly f empty. Completely freeing each superblock in turn will cause the superblock to first be released to the global heap and every subsequent free to a block in that superblock will therefore acquire the global heap lock. Luckily, this pathological case is highly unlikely to occur since it requires an improbable sequence of operations: the program must systematically free $(1 - f)$ of each superblock and then free every block in a superblock one at a time.

For the common case, Hoard will incur *very low* contention costs for any memory operation. This situation holds when the amount of live memory remains within the empty fraction of the maximum amount of memory allocated (and when all frees are local). Johnstone and Stefanović show in their empirical studies of allocation behavior that for nearly every program they analyzed, the memory in use tends to vary within a range that is within a fraction of total memory currently in use, and this amount often grows steadily [18, 32]. Thus, in the steady state case, Hoard incurs no contention, and in gradual growth, Hoard incurs low contention.

5. Experimental Results

In this section, we describe our experimental results. We performed experiments on uniprocessors and multiprocessors to demonstrate Hoard's speed, scalability, false sharing avoidance, and low fragmentation. We also show that these results are robust with respect to the choice of the empty fraction. The platform used is a dedicated 14-processor Sun Enterprise 5000 with 2GB of RAM and 400MHz UltraSparcs with 4 MB of level 2 cache, running Solaris 7. Except for the Barnes-Hut benchmark, all programs (including the allocators) were compiled using the GNU C++ compiler at the highest possible optimization level (-O6). We used GNU C++ instead of the vendor compiler (Sun Workshop compiler version 5.0) because we encountered errors when we used high optimization levels. In the experiments cited below, the size of a superblock S is 8K, the empty fraction f is $1/4$, the number of superblocks K that must be free for superblocks to be released is 4, and the base of the exponential for size classes b is 1.2 (bounding internal fragmentation to 1.2).

We compare Hoard (version 2.0.2) to the following single and multiple-heap memory allocators: *Solaris*, the default allocator provided with Solaris 7, *Ptmalloc* [9], the Linux allocator included in the GNU C library that extends a traditional allocator to use multiple heaps, and *MTmalloc*, a multiple heap allocator included with Solaris 7 for use with multithreaded parallel applications. (Section 6 includes extensive discussion of *Ptmalloc*, *MTmalloc*, and other concurrent allocators.) The latter two are the only publicly-available concurrent allocators of which we are aware for the Solaris platform (for example, *LKmalloc* is Microsoft proprietary). We use the Solaris allocator as the baseline for calculating speedups.

We use the single-threaded applications from Wilson and Johnstone, and Grunwald and Zorn [12, 19]: *espresso*, an optimizer for programmable logic arrays; *Ghostscript*, a PostScript interpreter; *LRUsim*, a locality analyzer, and *p2c*, a Pascal-to-C translator. We chose these programs because they are allocation-intensive and have

single-threaded benchmarks [12, 19]	
espresso	optimizer for programmable logic arrays
Ghostscript	PostScript interpreter
LRUsim	locality analyzer
p2c	Pascal-to-C translator
multithreaded benchmarks	
threadtest	each thread repeatedly allocates and then deallocates 100,000/P objects
shbench [26]	each thread allocates and randomly frees random-sized objects
Larson [22]	simulates a server: each thread allocates and deallocates objects, and then transfers some objects to other threads to be freed
active-false	tests active false sharing avoidance
passive-false	tests passive false sharing avoidance
BEMengine [7]	object-oriented PDE solver
Barnes-Hut [1, 2]	n-body particle solver

Table 1: Single- and multithreaded benchmarks used in this paper.

program	**runtime (sec)**		**change**
	Solaris	**Hoard**	
single-threaded benchmarks			
espresso	6.806	7.887	+15.9%
Ghostscript	3.610	3.993	+10.6%
LRUsim	1615.413	1570.488	-2.9%
p2c	1.504	1.586	+5.5%
multithreaded benchmarks			
threadtest	16.549	15.599	-6.1%
shbench	12.730	18.995	+49.2%
active-false	18.844	18.959	+0.6%
passive-false	18.898	18.955	+0.3%
BEMengine	678.30	614.94	-10.3%
Barnes-Hut	192.51	190.66	-1.0%
average			*+6.2%*

Table 2: Uniprocessor runtimes for single- and multithreaded benchmarks.

widely varying memory usage patterns. We used the same inputs for these programs as Wilson and Johnstone [19].

There is as yet no standard suite of benchmarks for evaluating multithreaded allocators. We know of no benchmarks that specifically stress multithreaded performance of server applications like web servers [1] and database managers. We chose benchmarks described in other papers and otherwise published (the *Larson* benchmark from Larson and Krishnan [22] and the *shbench* benchmark from MicroQuill, Inc. [26]), two multithreaded applications which include benchmarks (*BEMengine* [7] and *barnes-hut* [1, 2]), and wrote some microbenchmarks of our own to stress different aspects of memory allocation performance (*threadtest*, *active-false*, *passive-false*). Table 1 describes all of the benchmarks used in this paper. Table 4 includes their allocation behavior: fragmentation, maximum memory in use (U) and allocated (A), total memory requested, number of objects requested, and average object size.

5.1 Speed

Table 2 lists the uniprocessor runtimes for our applications when linked with Hoard and the Solaris allocator (each is the average of three runs; the variation between runs was negligible). On average, Hoard causes a slight increase in the runtime of these applications (6.2%), but this loss is primarily due to its performance on *shbench*. Hoard performs poorly on *shbench* because *shbench* uses a wide range of size classes but allocates very little memory (see Section 5.4 for more details). The longest-running application, *LRUsim*, runs almost 3% faster with Hoard. Hoard also performs well on *BEMengine* (10.3% faster than with the Solaris allocator), which allocates more memory than any of our other benchmarks (nearly 600MB).

5.2 Scalability

In this section, we present our experiments to measure scalability. We measure *speedup* with respect to the Solaris allocator. These applications vigorously exercise the allocators as revealed by the large difference between the maximum in use and the total memory requested (see Table 4).

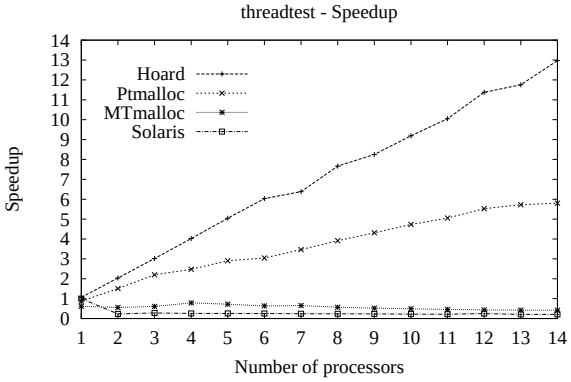
Figure 3 shows that Hoard matches or outperforms all of the allocators we tested. The Solaris allocator performs poorly overall because serial single heap allocators do not scale. *MTmalloc* often suffers from a centralized bottleneck. *Ptmalloc* scales well only when memory operations are fairly infrequent (the *Barnes-Hut* benchmark in Figure 3(d)); otherwise, its scaling peaks at around 6 processors. We now discuss each benchmark in turn.

In *threadtest*, t threads do nothing but repeatedly allocate and deallocate $100,000/t$ 8-byte objects (the threads do not synchronize or share objects). As seen in Figure 3(a), Hoard exhibits linear speedup, while the Solaris and *MTmalloc* allocators exhibit severe slowdown. For 14 processors, the Hoard version runs 278% faster than the *Ptmalloc* version. Unlike *Ptmalloc*, which uses a linked-list of heaps, Hoard does not suffer from a scalability bottleneck caused by a centralized data structure.

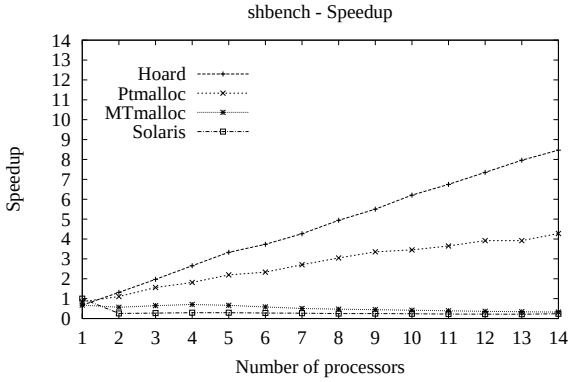
The *shbench* benchmark is available on MicroQuill's website and is shipped with the SmartHeap SMP product [26]. This benchmark is essentially a "stress test" rather than a realistic simulation of application behavior. Each thread repeatedly allocates and frees a number of randomly-sized blocks in random order, for a total of 50 million allocated blocks. The graphs in Figure 3(b) show that Hoard scales quite well, approaching linear speedup as the number of threads increases. The slope of the speedup line is less than ideal because the large number of different size classes hurts Hoard's raw performance. For 14 processors, the Hoard version runs 85% faster than the next best allocator (*Ptmalloc*). Memory usage in *shbench* remains within the empty fraction during the entire run so that Hoard incurs very low synchronization costs, while *Ptmalloc* again runs into its scalability bottleneck.

The intent of the *Larson* benchmark, due to Larson and Krishnan [22], is to simulate a workload for a server. A number of threads are repeatedly spawned to allocate and free 10,000 blocks ranging from 10 to 100 bytes 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 benchmark runs for 30 seconds and then reports the number of memory operations per second. Figure 3(c) shows that Hoard scales linearly, attaining nearly ideal speedup. For 14 processors, the Hoard version runs 18 times faster than the next best allocator, the *Ptmalloc* version. After an initial start-up phase, *Larson*

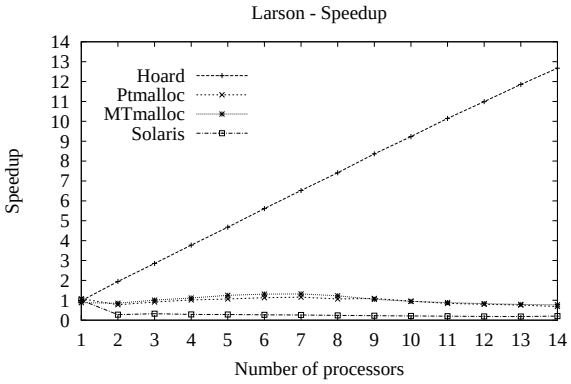
[1] Memory allocation becomes a bottleneck when most pages served are dynamically generated (Jim Davidson, personal communication). Unfortunately, the SPECweb99 benchmark [31] performs very few requests for completely dynamically-generated pages (0.5%), and most web servers exercise dynamic memory allocation only when generating dynamic content.



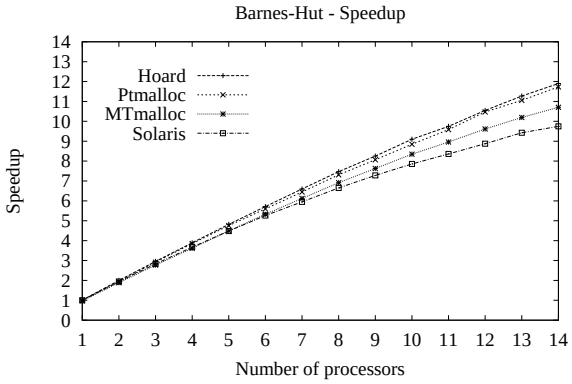
(a) The Threadtest benchmark.



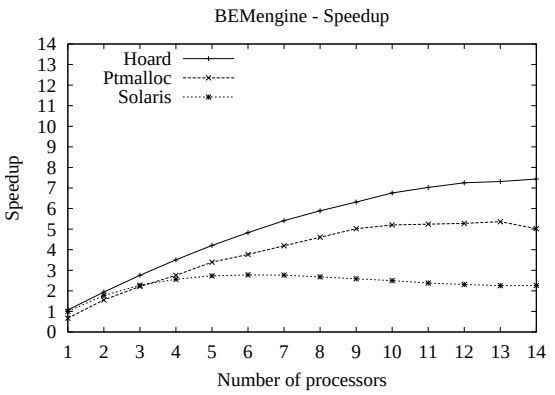
(b) The SmartHeap benchmark (*shbench*).



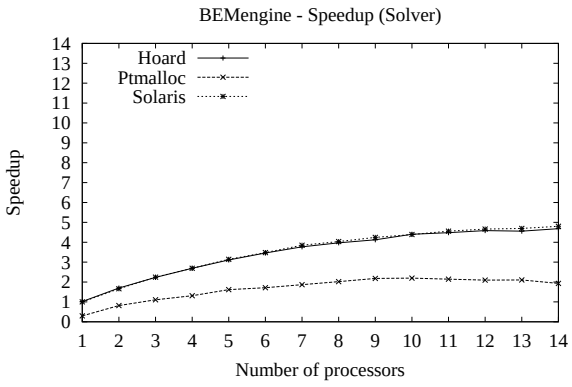
(c) Speedup using the Larson benchmark.



(d) Barnes-Hut speedup.



(e) *BEMengine* speedup. Linking with *MTmalloc* caused an exception to be raised.



(f) *BEMengine* speedup for the system solver only.

Figure 3: Speedup graphs.

remains within its empty fraction for most of the rest of its run (dropping below only a few times over a 30-second run and over 27 million mallocs) so that Hoard incurs very low synchronization costs. Despite the fact that *Larson* transfers many objects from one thread to another, Hoard performs quite well. All of the other allocators fail to scale at all, running slower on 14 processors than on one processor.

Barnes-Hut is a hierarchical n-body particle solver included with the Hood user-level multiprocessor threads library [1, 2], run on 32,768 particles for 20 rounds. This application performs a small amount of dynamic memory allocation during the tree-building phase. With 14 processors, all of the multiple-heap allocators provide a 10% performance improvement, increasing the speedup of the application from less than 10 to just above 12 (see Figure 3(d)). Hoard performs only slightly better than *Ptmalloc* in this case because this program does not exercise the allocator much. Hoard's performance is probably somewhat better simply because *Barnes-Hut* never drops below its empty fraction during its execution.

The *BEMengine* benchmark uses the solver engine from Coyote Systems' BEMSolver [7], a 2D/3D field solver that can solve electrostatic, magnetostatic and thermal systems. We report speedup for the three mostly-parallel parts of this code (equation registration, preconditioner creation, and the solver). Figure 3(e) shows that Hoard provides a significant runtime advantage over *Ptmalloc* and the Solaris allocator (*MTmalloc* caused the application to raise a fatal exception). During the first two phases of the program, the program's memory usage dropped below the empty fraction only 25 times over 50 seconds, leading to low synchronization overhead. This application causes *Ptmalloc* to exhibit pathological behavior that we do not understand, although we suspect that it derives from false sharing. During the execution of the solver phase of the computation, as seen in Figure 3(f), contention in the allocator is not an issue, and both Hoard and the Solaris allocator perform equally well.

5.3 False sharing avoidance

The *active-false* benchmark tests whether an allocator avoids actively inducing false sharing. Each thread allocates one small object, writes on it a number of times, and then frees it. The rate of memory allocation is low compared to the amount of work done, so this benchmark only tests contention caused by the cache coherence mechanism (cache ping-ponging) and not allocator contention. While Hoard scales linearly, showing that it avoids actively inducing false sharing, both *Ptmalloc* and *MTmalloc* only scale up to about 4 processors because they actively induce some false sharing. The Solaris allocator does not scale at all because it actively induces false sharing for nearly every cache line.

The *passive-false* benchmark tests whether an allocator avoids both passive and active false sharing by allocating a number of small objects and giving one to each thread, which immediately frees the object. The benchmark then continues in the same way as the *active-false* benchmark. If the allocator does not coalesce the pieces of the cache line initially distributed to the various threads, it passively induces false sharing. Figure 4(b) shows that Hoard scales nearly linearly; the gradual slowdown after 12 processors is due to program-induced bus traffic. Neither *Ptmalloc* nor *MTmalloc* avoid false sharing here, but the cause could be either active or passive false sharing.

In Table 3, we present measurements for our multithreaded benchmarks of the number of objects that could have been responsible for allocator-induced false sharing (i.e., those objects already in a superblock acquired from the global heap). In every case, when the per-processor heap acquired superblocks from the global heap, the superblocks were empty. These results demonstrate that Hoard successfully avoids allocator-induced false sharing.

program	falsely-shared objects
threadtest	0
shbench	0
Larson	0
BEMengine	0
Barnes-Hut	0

Table 3: Possible falsely-shared objects on 14 processors.

5.4 Fragmentation

We showed in Section 3.1 that Hoard has bounded blowup. In this section, we measure Hoard's average case fragmentation. We use a number of single- and multithreaded applications to evaluate Hoard's average-case fragmentation.

Collecting fragmentation information for multithreaded applications is problematic because fragmentation is a global property. Updating the maximum memory in use and the maximum memory allocated would serialize all memory operations and thus seriously perturb allocation behavior. We cannot simply use the maximum memory in use for a serial execution because a parallel execution of a program may lead it to require much more memory than a serial execution.

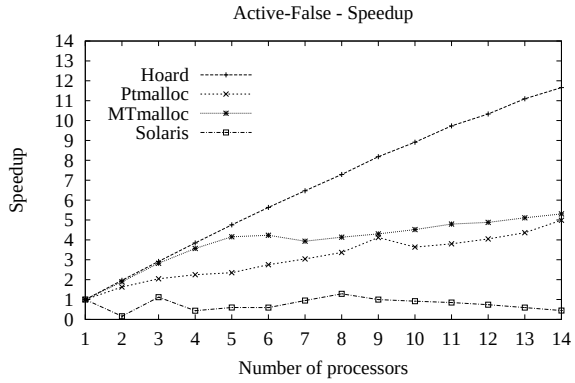
We solve this problem by collecting traces of memory operations and processing these traces off-line. We modified Hoard so that (when collecting traces) each per-processor heap records every memory operation along with a timestamp (using the SPARC high-resolution timers via gethrtime()) into a memory-mapped buffer and writes this trace to disk upon program termination. We then merge the traces in timestamp order to build a complete trace of memory operations and process the resulting trace to compute maximum memory allocated and required. Collecting these traces results in nearly a threefold slowdown in memory operations but does not excessively disturb their parallelism, so we believe that these traces are a faithful representation of the fragmentation induced by Hoard.

Single-threaded Applications

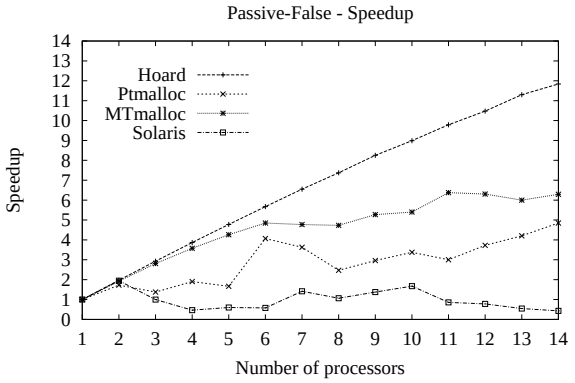
We measured fragmentation for the single-threaded benchmarks. We follow Wilson and Johnstone [19] and report memory allocated without counting overhead (like per-object headers) to focus on the allocation *policy* rather than the *mechanism*. Hoard's fragmentation for these applications is between 1.05 and 1.2, except for *espresso*, which consumes 46% more memory than it requires. *Espresso* is an unusual program since it uses a large number of different size classes for a small amount of memory required (less than 300K), and this behavior leads Hoard to waste space within each 8K superblock.

Multithreaded Applications

Table 4 shows that the fragmentation results for the multithreaded benchmarks are generally quite good, ranging from nearly no fragmentation (1.02) for *BEMengine* to 1.24 for *threadtest*. The anomaly is *shbench*. This benchmark uses a large range of object sizes, randomly chosen from 8 to 100, and many objects remain live for the duration of the program (470K of its maximum 550K objects remain in use at the end of the run cited here). These unfreed objects are randomly scattered across superblocks, making it impossible to recycle them for different size classes. This extremely random behavior is not likely to be representative of real programs [19] but it does show that Hoard's method of maintaining one size class per



(a) Speedup for the *active-false* benchmark, which fails to scale with memory allocators that *actively* induce false sharing.



(b) Speedup for the *passive-false* benchmark, which fails to scale with memory allocators that *passively* or *actively* induce false sharing.

Figure 4: Speedup graphs that exhibit the effect of allocator-induced false sharing.

Benchmark applications	Hoard fragmentation (A/U)	max in use (U)	max allocated (A)	total memory requested	# objects requested	average object size
single-threaded benchmarks						
espresso	1.47	284,520	417,032	110,143,200	1,675,493	65.7378
Ghostscript	1.15	1,171,408	1,342,240	52,194,664	566,542	92.1285
LRUsim	1.05	1,571,176	1,645,856	1,588,320	39,109	40.6126
p2c	1.20	441,432	531,912	5,483,168	199,361	27.5037
multithreaded benchmarks						
threadtest	1.24	1,068,864	1,324,848	80,391,016	9,998,831	8.04
shbench	3.17	556,112	1,761,200	1,650,564,600	12,503,613	132.00
Larson	1.22	8,162,600	9,928,760	1,618,188,592	27,881,924	58.04
BEMengine	1.02	599,145,176	613,935,296	4,146,087,144	18,366,795	225.74
Barnes-Hut	1.18	11,959,960	14,114,040	46,004,408	1,172,624	39.23

Table 4: Hoard fragmentation results and application memory statistics. We report fragmentation statistics for 14-processor runs of the multithreaded programs. All units are in bytes.

superblock can yield poor memory efficiency for certain behaviors, although Hoard still attains good scalable performance for this application (see Figure 3(b)).

5.5 Sensitivity Study

We also examined the effect of changing the empty fraction on runtime and fragmentation for the multithreaded benchmarks. Because superblocks are returned to the global heap (for reuse by other threads) when the heap crosses the emptiness threshold, the empty fraction affects both synchronization and fragmentation. We varied the empty fraction from $1/8$ to $1/2$ and saw very little change in runtime and fragmentation. We chose this range to exercise the tension between increased (worst-case) fragmentation and synchronization costs. The only benchmark which is substantially affected by these changes in the empty fraction is the *Larson* benchmark, whose fragmentation increases from 1.22 to 1.61 for an empty fraction of $1/2$. Table 5 presents the runtime for these programs on 14 processors (we report the number of memory operations per second for the Larson benchmark, which runs for 30 seconds), and Table 6 presents the fragmentation results. Hoard's runtime is robust with respect to changes in the empty fraction because programs tend to reach a steady state in memory usage and stay within even as small an empty fraction as $1/8$, as described in Section 4.2.

program	runtime (sec)		
	$f = 1/8$	$f = 1/4$	$f = 1/2$
threadtest	1.27	1.28	1.19
shbench	1.45	1.50	1.44
BEMengine	86.85	87.49	88.03
Barnes-Hut	16.52	16.13	16.41
	throughput (memory ops/sec)		
Larson	4,407,654	4,416,303	4,352,163

Table 5: Runtime on 14 processors using Hoard with different empty fractions.

6. Related Work

While dynamic storage allocation is one of the most studied topics in computer science, there has been relatively little work on concurrent memory allocators. In this section, we place past work into a taxonomy of memory allocator algorithms and compare each to Hoard. We address the blowup and allocator-induced false sharing characteristics of each of these allocator algorithms and compare them to Hoard.

program	fragmentation		
	$f = 1/8$	$f = 1/4$	$f = 1/2$
threadtest	1.22	1.24	1.22
shbench	3.17	3.17	3.16
Larson	1.22	1.22	1.61
BEMengine	1.02	1.02	1.02
Barnes-Hut	1.18	1.18	1.18

Table 6: Fragmentation on 14 processors using Hoard with different empty fractions.

6.1 Taxonomy of Memory Allocator Algorithms

Our taxonomy consists of the following five categories:

Serial single heap. Only one processor may access the heap at a time (Solaris, Windows NT/2000 [21]).

Concurrent single heap. Many processors may simultaneously operate on one shared heap ([5, 16, 17, 13, 14]).

Pure private heaps. Each processor has its own heap (STL [30], Cilk [6]).

Private heaps with ownership. Each processor has its own heap, but memory is always returned to its "owner" processor (*MTmalloc*, *Ptmalloc* [9], *LKmalloc* [22]).

Private heaps with thresholds. Each processor has its own heap which can hold a limited amount of free memory (DYNIX kernel allocator [25], Vee and Hsu [37], Hoard).

Below we discuss these single and multiple-heap algorithms, focusing on the false sharing and blowup characteristics of each.

Single Heap Allocation

Serial single heap allocators often exhibit extremely low fragmentation over a wide range of real programs [19] and are quite fast [23]. Since they typically protect the heap with a single lock which serializes memory operations and introduces contention, they are inappropriate for use with most parallel multithreaded programs. In multithreaded programs, contention for the lock prevents allocator performance from scaling with the number of processors. Most modern operating systems provide such memory allocators in the default library, including Solaris and IRIX. Windows NT/2000 uses 64-bit atomic operations on freelists rather than locks [21] which is also unscalable because the head of each freelist is a central bottleneck[2]. These allocators all actively induce false sharing.

Concurrent single heap allocation implements the heap as a concurrent data structure, such as a concurrent B-tree [10, 11, 13, 14, 16, 17] or a freelist with locks on each free block [5, 8, 34]. This approach reduces to a serial single heap in the common case when most allocations are from a small number of object sizes. Johnstone and Wilson show that for every program they examined, the vast majority of objects allocated are of only a few sizes [18]. Each memory operation on these structures requires either time linear in the number of free blocks or $O(\log C)$ time, where C is the number of *size classes* of allocated objects. A size class is a range of object sizes that are grouped together (e.g., all objects between 32 and 36 bytes are treated as 36-byte objects). Like serial single heaps, these allocators actively induce false sharing. Another problem with these allocators is that they make use of many locks or atomic update operations (e.g., compare-and-swap), which are quite expensive.

State-of-the-art serial allocators are so well engineered that most memory operations involve only a handful of instructions [23]. An *uncontended* lock acquire and release accounts for about half of the total runtime of these memory operations. In order to be competitive, a memory allocator can only acquire and release at most two locks in the common case, or incur three atomic operations. Hoard requires only one lock for each malloc and two for each free and each memory operation takes constant (amortized) time (see Section 3.4).

Multiple-Heap Allocation

We describe three categories of allocators which all use multiple-heaps. The allocators assign threads to heaps either by assigning one heap to every thread (using thread-specific data) [30], by using a currently unused heap from a collection of heaps [9], round-robin heap assignment (as in *MTmalloc*, provided with Solaris 7 as a replacement allocator for multithreaded applications), or by providing a mapping function that maps threads onto a collection of heaps (*LKmalloc* [22], Hoard). For simplicity of exposition, we assume that there is exactly one thread bound to each processor and one heap for each of these threads.

STL's (Standard Template Library) *pthread_alloc*, Cilk 4.1, and many ad hoc allocators use *pure private heaps* allocation [6, 30]. Each processor has its own per-processor heap that it uses for every memory operation (the allocator mallocs from its heap and frees to its heap). Each per-processor heap is "purely private" because each processor never accesses any other heap for any memory operation. After one thread allocates an object, a second thread can free it; in pure private heaps allocators, this memory is placed in the second thread's heap. Since parts of the same cache line may be placed on multiple heaps, pure private-heaps allocators passively induce false sharing. Worse, pure private-heaps allocators exhibit unbounded memory consumption given a producer-consumer allocation pattern, as described in Section 2.2. Hoard avoids this problem by returning freed blocks to the heap that owns the superblocks they belong to.

Private heaps with ownership returns free blocks to the heap that allocated them. This algorithm, used by *MTmalloc*, *Ptmalloc* [9] and *LKmalloc* [22], yields $O(P)$ blowup, whereas Hoard has $O(1)$ blowup. Consider a round-robin style producer-consumer program: each processor i allocates K blocks and processor $(i + 1) \bmod P$ frees them. The program requires only K blocks but the allocator will allocate $P * K$ blocks (K on all P heaps). *Ptmalloc* and *MTmalloc* can actively induce false sharing (different threads may allocate from the same heap). *LKmalloc*'s permanent assignment of large regions of memory to processors and its immediate return of freed blocks to these regions, while leading to $O(P)$ blowup, should have the advantage of eliminating allocator-induced false sharing, although the authors did not explicitly address this issue. Hoard explicitly takes steps to reduce false sharing, although it cannot avoid it altogether, while maintaining $O(1)$ blowup.

Both *Ptmalloc* and *MTmalloc* also suffer from scalability bottlenecks. In *Ptmalloc*, each malloc chooses the first heap that is not currently in use (caching the resulting choice for the next attempt). This heap selection strategy causes substantial bus traffic which limits *Ptmalloc*'s scalability to about 6 processors, as we show in Section 5. *MTmalloc* performs round-robin heap assignment by maintaining a "nextHeap" global variable that is updated by every call to malloc. This variable is a source of contention that makes *MTmalloc* unscalable and actively induces false sharing. Hoard has no centralized bottlenecks except for the global heap, which is not a

[2]The Windows 2000 allocator and some of Iyengar's allocators use one freelist for each object size or range of sizes [13, 14, 21]

Allocator algorithm	fast?	scalable?	avoids false sharing?	blowup
serial single heap	yes	no	no	$O(1)$
concurrent single heap	no	maybe	no	$O(1)$
pure private heaps	yes	yes	no	unbounded
private heaps w/ownership				
Ptmalloc [9]	yes	yes	no	$O(P)$
MTmalloc	yes	no	no	$O(P)$
LKmalloc [22]	yes	yes	yes	$O(P)$
private heaps w/thresholds				
Vee and Hsu, DYNIX [25, 37]	yes	yes	no	$O(1)$
Hoard	**yes**	**yes**	**yes**	$O(1)$

Table 7: A taxonomy of memory allocation algorithms discussed in this paper.

frequent source of contention for reasons described in Section 4.2.

The DYNIX kernel memory allocator by McKenney and Slingwine [25] and the single object-size allocator by Vee and Hsu [37] employ a *private heaps with thresholds* algorithm. These allocators are efficient and scalable because they move large blocks of memory between a hierarchy of per-processor heaps and heaps shared by multiple processors. When a per-processor heap has more than a certain amount of free memory (the threshold), some portion of the free memory is moved to a shared heap. This strategy also bounds blowup to a constant factor, since no heap may hold more than some fixed amount of free memory. The mechanisms that control this motion and the units of memory moved by the DYNIX and Vee and Hsu allocators differ significantly from those used by Hoard. Unlike Hoard, both of these allocators passively induce false sharing by making it very easy for pieces of the same cache line to be recycled. As long as the amount of free memory does not exceed the threshold, pieces of the same cache line spread across processors will be repeatedly reused to satisfy memory requests. Also, these allocators are forced to synchronize every time the threshold amount of memory is allocated or freed, while Hoard can avoid synchronization altogether while the emptiness of per-processor heaps is within the empty fraction. On the other hand, these allocators do avoid the two-fold slowdown that can occur in the worst-case described for Hoard in Section 4.2.

Table 7 presents a summary of the above allocator algorithms, along with their speed, scalability, false sharing and blowup characteristics. As can be seen from the table, the algorithms closest to Hoard are Vee and Hsu, DYNIX, and *LKmalloc*. The first two fail to avoid passively-induced false sharing and are forced to synchronize with a global heap after each threshold amount of memory is consumed or freed, while Hoard avoids false sharing and is not required to synchronize until the emptiness threshold is crossed or when a heap does not have sufficient memory. *LKmalloc* has similar synchronization behavior to Hoard and avoids allocator-induced false sharing, but has $O(P)$ blowup.

7. Future Work

Although the hashing method that we use has so far proven to be an effective mechanism for assigning threads to heaps, we plan to develop an efficient method that can adapt to the situation when two concurrently-executing threads map to the same heap.

While we believe that Hoard improves program locality in various ways, we have yet to quantitatively measure this effect. We plan to use both cache and page-level measurement tools to evaluate and improve Hoard's effect on program-level locality.

We are also looking at ways to remove the one size class per superblock restriction. This restriction is responsible for increased fragmentation and a decline in performance for programs which allocate objects from a wide range of size classes, like *espresso* and *shbench*.

Finally, we are investigating ways of improving performance of Hoard on cc/NUMA architectures. Because the unit of cache coherence on these architectures is an entire page, Hoard's mechanism of coalescing to page-sized superblocks appears to be very important for scalability. Our preliminary results on an SGI Origin 2000 show that Hoard scales to a substantially larger number of processors, and we plan to report these results in the future.

8. Conclusion

In this paper, we have introduced the Hoard memory allocator. Hoard improves on previous memory allocators by simultaneously providing four features that are important for scalable application performance: speed, scalability, false sharing avoidance, and low fragmentation. Hoard's novel organization of per-processor and global heaps along with its discipline for moving superblocks across heaps enables Hoard to achieve these features and is the key contribution of this work. Our analysis shows that Hoard has provably bounded blowup and low expected case synchronization. Our experimental results on eleven programs demonstrate that in practice Hoard has low fragmentation, avoids false sharing, and scales very well. In addition, we show that Hoard's performance and fragmentation are robust with respect to its primary parameter, the empty fraction. Since scalable application performance clearly requires scalable architecture and runtime system support, Hoard thus takes a key step in this direction.

9. Acknowledgements

Many thanks to Brendon Cahoon, Rich Cardone, Scott Kaplan, Greg Plaxton, Yannis Smaragdakis, and Phoebe Weidmann for valuable discussions during the course of this work and input during the writing of this paper. Thanks also to Martin Bächtold, Trey Boudreau, Robert Fleischman, John Hickin, Paul Larson, Kevin Mills, and Ganesan Rajagopal for their contributions to helping to improve and port Hoard, and to Ben Zorn and the anonymous reviewers for helping to improve this paper.

Hoard is publicly available at `http://www.hoard.org` for a variety of platforms, including Solaris, IRIX, AIX, Linux, and Windows NT/2000.

10. References

[1] U. Acar, E. Berger, R. Blumofe, and D. Papadopoulos. Hood: A threads library for multiprogrammed multiprocessors. http://www.cs.utexas.edu/users/hood, Sept. 1999.

[2] J. Barnes and P. Hut. A hierarchical $O(N \log N)$ force-calculation algorithm. *Nature*, 324:446–449, 1986.

[3] bCandid.com, Inc. http://www.bcandid.com.

[4] E. D. Berger and R. D. Blumofe. Hoard: A fast, scalable, and memory-efficient allocator for shared-memory multiprocessors. Technical Report UTCS-TR99-22, The University of Texas at Austin, 1999.

[5] B. Bigler, S. Allan, and R. Oldehoeft. Parallel dynamic storage allocation. *International Conference on Parallel Processing*, pages 272–275, 1985.

[6] R. D. Blumofe and C. 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, Nov. 1994.

[7] Coyote Systems, Inc. http://www.coyotesystems.com.

[8] C. S. Ellis and T. J. Olson. Algorithms for parallel memory allocation. *International Journal of Parallel Programming*, 17(4):303–345, 1988.

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

[10] A. Gottlieb and J. Wilson. Using the buddy system for concurrent memory allocation. Technical Report System Software Note 6, Courant Institute, 1981.

[11] A. Gottlieb and J. Wilson. Parallelizing the usual buddy algorithm. Technical Report System Software Note 37, Courant Institute, 1982.

[12] D. Grunwald, B. Zorn, and R. Henderson. Improving the cache locality of memory allocation. In R. Cartwright, editor, *Proceedings of the Conference on Programming Language Design and Implementation*, pages 177–186, New York, NY, USA, June 1993. ACM Press.

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

[14] A. K. Iyengar. Parallel dynamic storage allocation algorithms. In *Fifth IEEE Symposium on Parallel and Distributed Processing*. IEEE Press, 1993.

[15] T. Jeremiassen and S. Eggers. Reducing false sharing on shared memory multiprocessors through compile time data transformations. In *ACM Symposium on Principles and Practice of Parallel Programming*, pages 179–188, July 1995.

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

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

[18] M. S. Johnstone. *Non-Compacting Memory Allocation and Real-Time Garbage Collection*. PhD thesis, University of Texas at Austin, Dec. 1997.

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

[20] K. Kennedy and K. S. McKinley. Optimizing for parallelism and data locality. In *Proceedings of the Sixth International Conference on Supercomputing*, pages 323–334, Distributed Computing, July 1992.

[21] M. R. Krishnan. Heap: Pleasures and pains. Microsoft Developer Newsletter, Feb. 1999.

[22] P. Larson and M. Krishnan. Memory allocation for long-running server applications. In *ISMM*, Vancouver, B.C., Canada, 1998.

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

[24] B. Lewis. `comp.programming.threads` FAQ. http://www.lambdacs.com/newsgroup/FAQ.html.

[25] P. E. McKenney and J. Slingwine. Efficient kernel memory allocation on shared-memory multiprocessor. In USENIX Association, editor, *Proceedings of the Winter 1993 USENIX Conference: January 25–29, 1993, San Diego, California, USA*, pages 295–305, Berkeley, CA, USA, Winter 1993. USENIX.

[26] MicroQuill, Inc. http://www.microquill.com.

[27] MySQL, Inc. The mysql database manager. http://www.mysql.org.

[28] G. J. Narlikar and G. E. Blelloch. Space-efficient scheduling of nested parallelism. *ACM Transactions on Programming Languages and Systems*, 21(1):138–173, January 1999.

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

[30] SGI. The standard template library for c++: Allocators. http://www.sgi.com/Technology/STL/Allocators.html.

[31] Standard Performance Evaluation Corporation. SPECweb99. http://www.spec.org/osg/web99/.

[32] D. Stefanović. *Properties of Age-Based Automatic Memory Reclamation Algorithms*. PhD thesis, Department of Computer Science, University of Massachusetts, Amherst, Massachusetts, Dec. 1998.

[33] D. Stein and D. Shah. Implementing lightweight threads. In *Proceedings of the 1992 USENIX Summer Conference*, pages 1–9, 1992.

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

[35] Time-Warner/AOL, Inc. AOLserver 3.0. http://www.aolserver.com.

[36] J. Torrellas, M. S. Lam, and J. L. Hennessy. False sharing and spatial locality in multiprocessor caches. *IEEE Transactions on Computers*, 43(6):651–663, 1994.

[37] V.-Y. Vee and W.-J. 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.