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 programs such as web servers, database managers, news servers, and scientific applications are becoming increasingly prevalent. For these C and C++ 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 *blowup* in memory consumption when confronted with a producer-consumer pattern of object allocation and freeing. This blowup can increase memory consumption by a factor of P (the number of processors) or lead to unbounded memory consumption. Such pathological behavior can cause premature program termination by exhausting all available swap space.

This paper introduces Hoard, a fast, highly scalable allocator that avoids false sharing and blowup. Hoard is the first allocator to simultaneously solve the above problems. Hoard combines one global heap and P per-processor heaps with a novel discipline that provably bounds blowup and has near zero 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 [33], database managers [26], 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 key 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 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 2) exhibit a special kind of fragmentation that we call ***blowup***. Intuitively, blowup is the increase in memory consumption caused when an allocator systematically makes memory unavailable for future memory requests (we define blowup formally in Section 4). As we show in Section 2.2, the common producer-consumer programming idiom causes 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 major 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 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 has very low synchronization costs for most programs. This algorithm also avoids false sharing, because pieces of a cache line are available for reuse by only one processor (by a subsequent call to malloc). 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

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.

This paper has been submitted for publication.

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 2.1, we explain in detail the issues of blowup and allocator-induced false sharing. In Section 2, we classify previous work into a taxonomy of memory allocators and show why no previous work solves all the speed, scalability, false sharing and fragmentation problems described above. 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 of its key parameter. Finally, we discuss future directions for this research in Section 6, and conclude in Section 7.

2 Related Work

In this section, we first focus special attention on the key 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 ignored in the memory allocation literature. We then place past work into a taxonomy of memory allocator algorithms and compare each to Hoard.

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, 34]. 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, 8 byte objects would be padded to 64 bytes on a SPARC) and 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. 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 will 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. 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 these pieces, which can then lead to false sharing.

2.2 Blowup

Many of the allocators described in Section 2.3 suffer from blowup. While Hoard keeps blowup to a constant factor, the blowup of many existing concurrent allocators is either *unbounded* [6, 29] (memory consumption grows without bound while the memory required is fixed) or can grow linearly with P, the number of processors [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, the literature does 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. Since 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 [27, 6]. 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 Ps. If the allocator multiplies this consumption by another factor of P, then memory consumption increases by P^2.

2.3 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 [29], 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 (Vee and Hsu [35], Hoard).

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

2.3.1 Single Heap Allocation

Serial single heap allocators, can exhibit extremely low fragmentation over a wide range of real programs [19] and are quite fast [23]. However, they are inappropriate for most parallel multithreaded programs since they typically protect the heap with a single lock which serializes memory operations and introduces contention. 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 (see Note 1). 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, 32].[1] Each memory operation on these structures requires 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.

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

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 [35]	yes	yes	no	$O(1)$
Hoard	**yes**	**yes**	**yes**	$O(1)$

Table 1: A taxonomy of memory allocation algorithms discussed in this paper.

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.

2.3.2 Multiple-Heap Allocation

In the discussion below, we describe three categories of allocators which all use multiple-heaps. The allocators assign heaps to threads either by assigning one heap to every thread (using thread-specific data) [29], by using a currently unused heap from a collection of heaps [9], round-robin heap assignment (as in *MTmalloc*), 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, 29]. Each processor has its own per-processor heap that it uses for every memory operation (the allocator malloc's from its heap and free's 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 a pure private heaps allocator, 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, a pure private-heaps allocator can exhibit unbounded memory consumption, as described in Section 2.2.

Private heaps with ownership returns free blocks to the heap that allocated them. This algorithm, used by *MTmalloc*, Gloger's *Ptmalloc* [9] and Larson and Krishnan's *LKmalloc* [22], yields $O(P)$ 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 PK blocks (K on all P heaps). Gloger's allocator and *MTmalloc* can actively induce false sharing (different threads may allocate from the same heap). We believe that Larson and Krishnan's algorithm avoids allocator-induced false sharing, although they did not explicitly address this issue. 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 limits *Ptmalloc*'s scalability to about 6 processors, as we show in Section 5. *MTmalloc*, provided with Solaris 7 as a replacement allocator for multithreaded applications, 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.

The DYNIX kernel memory allocator by McKenney and Slingwine [24] and the single object-size allocator by Vee and Hsu [35] 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 allowing pieces of the same cache line to be recycled.

Table 1 presents a summary of allocator algorithms, along with their speed, scalability, false sharing and blowup characteristics. Hoard is the only one that solves all four problems.

3 The Hoard Memory Allocator

In this section, we describe Hoard in detail. Hoard can be viewed as a new kind of private heaps with thresholds allocator that avoids false sharing and that can trade increased (but bounded) memory consumption for reduced, or even zero, synchronization costs.

Hoard augments per-processor heaps with a *global heap* that every thread may access (similar to Vee and Hsu [35]). Each thread can access only its heap and the global heap. We designate heap 0 as the global heap and heaps 1 through $2P$ as the per-processor heaps. We use $2P$ heaps to decrease the probability that concurrently-executing threads hash to the same heap. We use a simple hash function to map thread id's to per-processor heaps (on Solaris, we hash on the light-weight process id). We need 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.

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. All superblocks are the same 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 (wasted space within an object) 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.