Scalable Lock-Free Dynamic Memory Allocation

Maged M. Michael
IBM Thomas J. Watson Research Center
P.O. Box 218, Yorktown Heights, NY 10598, USA
magedm@us.ibm.com

ABSTRACT

Dynamic memory allocators (malloc/free) rely on mutual exclusion locks for protecting the consistency of their shared data structures under multithreading. The use of locking has many disadvantages with respect to performance, availability, robustness, and programming flexibility. A lock-free memory allocator guarantees progress regardless of whether some threads are delayed or even killed and regardless of scheduling policies. This paper presents a completely lock-free memory allocator. It uses only widely-available operating system support and hardware atomic instructions. It offers guaranteed availability even under arbitrary thread termination and crash-failure, and it is immune to deadlock regardless of scheduling policies, and hence it can be used even in interrupt handlers and real-time applications without requiring special scheduler support. Also, by leveraging some high-level structures from Hoard, our allocator is highly scalable, limits space blowup to a constant factor, and is capable of avoiding false sharing. In addition, our allocator allows finer concurrency and much lower latency than Hoard. We use PowerPC shared memory multiprocessor systems to compare the performance of our allocator with the default AIX 5.1 libc malloc, and two widely-used multithread allocators, Hoard and Ptmalloc. Our allocator outperforms the other allocators in virtually all cases and often by substantial margins, under various levels of parallelism and allocation patterns. Furthermore, our allocator also offers the lowest contention-free latency among the allocators by significant margins.

Categories and Subject Descriptors: D.1.3 [**Programming Techniques**]: Concurrent Programming; D.3.3 [**Programming Languages**]: Language Constructs and Features—*dynamic storage management*; D.4.1 [**Operating Systems**]: Process Management—*concurrency, deadlocks, synchronization, threads.*

General Terms: Algorithms, Performance, Reliability.

Keywords: malloc, lock-free, async-signal-safe, availability.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
PLDI'04, June 9–11, 2004, Washington, DC, USA.
Copyright 2004 ACM 1-58113-807-5/04/0006 ...$5.00.

1. INTRODUCTION

Dynamic memory allocation functions, such as malloc and free, are heavily used by a wide range of important multithreaded applications, from commercial database and web servers to data mining and scientific applications. In order to be safe under multithreading (MT-safe), current allocators employ mutual exclusion locking in a variety of ways, ranging from the use of a single lock wrapped around single-thread malloc and free, to the distributed use of locks in order to allow more concurrency and higher scalability. The use of locking causes many problems and limitations with respect to performance, availability, robustness, and programming flexibility.

A desirable but challenging alternative approach for achieving MT-safety is lock-free synchronization. A shared object is lock-free (nonblocking) if it guarantees that whenever a thread executes some finite number of steps, at least one operation on the object by some thread must have made progress during the execution of these steps. Lock-free synchronization implies several inherent advantages:

Immunity to deadlock: By definition, a lock-free object must be immune to deadlock and livelock. Therefore, it is much simpler to design deadlock-free systems when all or some of their components are lock-free.

Async-signal-safety: Due to the use of locking in current implementations of malloc and free, they are not considered *async-signal-safe* [9], i.e., signal handlers are prohibited from using them. The reason for this prohibition is that if a thread receives a signal while holding a user-level lock in the allocator, and if the signal handler calls the allocator, and in the process it must acquire the same lock held by the interrupted thread, then the allocator becomes deadlocked due to circular dependence. The signal handler waits for the interrupted thread to release the lock, while the thread cannot resume until the signal handler completes. Masking interrupts or using kernel-assisted locks in malloc and free is too costly for such heavily-used functions. In contrast, a completely lock-free allocator is capable of being async-signal-safe without incurring any performance cost.

Tolerance to priority inversion: Similarly, in real-time applications, user-level locking is susceptible to deadlock due to priority inversion. That is, a high priority thread can be waiting for a user-level lock to be released by a lower priority thread that will not be scheduled until the high priority thread completes its task. Lock-free synchronization guarantees progress regardless of scheduling policies.

Kill-tolerant availability: A lock-free object must be immune to deadlock even if any number of threads are killed while operating on it. Accordingly, a lock-free object must offer guaranteed availability regardless of arbitrary thread

```
CAS(addr,expval,newval) atomically do
  if (*addr == expval) {
    *addr = newval;
    return true;
  } else
    return false;
```

Figure 1: Compare-and-Swap.

```
AtomicInc(addr)
  do {
    oldval = *addr;
    newval = oldval+1;
  } until CAS(addr,oldval,newval);
```

Figure 2: Atomic increment using CAS.

termination or crash-failure. This is particularly useful for servers that require a high level of availability, but can tolerate the infrequent loss of tasks or servlets that may be killed by the server administrator in order to relieve temporary resource shortages.

Preemption-tolerance: When a thread is preempted while holding a mutual exclusion lock, other threads waiting for the same lock either spin uselessly, possibly for the rest of their time slices, or have to pay the performance cost of yielding their processors in the hope of giving the lock holder an opportunity to complete its critical section. Lock-free synchronization offers preemption-tolerant performance, regardless of arbitrary thread scheduling.

It is clear that it is desirable for memory allocators to be completely lock-free. The question is how, and more importantly, how to be completely lock-free and (1) offer good performance competitive with the best lock-based allocators (i.e., low latency, scalability, avoiding false sharing, constant space blowup factor, and robustness under contention and preemption), (2) using only widely-available hardware and OS support, and (3) without making trivializing assumptions that make lock-free progress easy, but result in unacceptable memory consumption or impose unreasonable restrictions.

For example, it is trivial to design a wait-free allocator with pure per-thread private heaps. That is, each thread allocates from its own heap and also frees blocks to its own heap. However, this is hardly an acceptable general-purpose solution, as it can lead to unbounded memory consumption (e.g., under a producer-consumer pattern [3]), even when the program's memory needs are in fact very small. Other unacceptable characteristics include the need for initializing large parts of the address space, putting an artificial limit on the total size or number of allocatable dynamic blocks, or restricting beforehand regions of the address to specific threads or specific block sizes. An acceptable solution must be general-purpose and space efficient, and should not impose artificial limitations on the use of the address space.

In this paper we present a completely lock-free allocator that offers excellent performance, uses only widely-available hardware and OS support, and is general-purpose.

For constructing our lock-free allocator and with only the simple atomic instructions supported on current mainstream processor architectures as our memory access tools, we break down malloc and free into fine atomic steps, and organize the allocator's data structures such that if any thread is delayed arbitrarily (or even killed) at any point, then any other thread using the allocator will be able to determine enough of the state of the allocator to proceed with its own operation without waiting for the delayed thread to resume.

By leveraging some high-level structures from Hoard [3], a scalable lock-based allocator, we achieve concurrency between operations on multiple processors, avoid inducing false sharing, and limit space blowup to a constant factor. In addition, our allocator uses a simpler and finer grained organization that allows more concurrency and lower latency than Hoard.

We use POWER3 and POWER4 shared memory multiprocessors to compare the performance of our allocator with the default AIX 5.1 libc malloc, and two widely-used lock-based allocators with mechanisms for scalable performance, Hoard [3] and Ptmalloc [6]. The experimental performance results show that not only is our allocator competitive with some of the best lock-based allocators, but also that it outperforms them, and often by substantial margins, in virtually all cases including under various levels of parallelism and various sharing patterns, and also offers the lowest contention-free latency.

The rest of the paper is organized as follows. In Section 2, we discuss atomic instructions and related work. Section 3 describes the new allocator in detail. Section 4 presents our experimental performance results. We conclude the paper with Section 5.

2. BACKGROUND

2.1 Atomic Instructions

Current mainstream processor architectures support either Compare-and-Swap (CAS) or the pair Load-Linked and Store-Conditional (LL/SC). Other weaker instructions, such as Fetch-and-Add and Swap, may be supported, but in any case they are easily implemented using CAS or LL/SC.

CAS was introduced on the IBM System 370 [8]. It is supported on Intel (IA-32 and IA-64) and Sun SPARC architectures. In its simplest form, it takes three arguments: the address of a memory location, an expected value, and a new value. If the memory location is found to hold the expected value, the new value is written to it, atomically. A Boolean return value indicates whether the write occurred. If it returns true, it is said to succeed. Otherwise, it is said to fail. Figure 1 shows the semantics of CAS.

LL and SC are supported on the PowerPC, MIPS, and Alpha architectures. LL takes one argument: the address of a memory location, and returns its contents. SC takes two arguments: the address of a memory location and a new value. Only if no other thread has written the memory location since the current thread last read it using LL, the new value is written to the memory location, atomically. A Boolean return value indicates whether the write occurred. Similar to CAS, SC is said to succeed or fail if it returns true or false, respectively. For architectural reasons, current architectures that support LL/SC do not allow the nesting or interleaving of LL/SC pairs, and infrequently allow SC to fail spuriously, even if the target location was never written since the last LL. These spurious failures happen, for example, if the thread was preempted or a different location in the same cache line was written by another processor.

For generality, we present the algorithms in this paper using CAS. If LL/SC are supported rather than CAS, then `CAS(addr,expval,newval)` can be simulated in a lock-free

manner as follows: `{do {if (LL(addr)!=expval) return false;} until SC(addr,newval); return true;}`.

Support for CAS and restricted LL/SC on aligned 64-bit blocks is available on both 32-bit and 64-bit architectures, e.g., CMPXCHG8 on IA-32. However, support for CAS or LL/SC on wider block sizes is generally not available even on 64-bit architectures. Therefore, we focus our presentation of the algorithms on 64-bit mode, as it is the more challenging case while 32-bit mode is simpler.

For a very simple example of lock-free synchronization, Figure 2 shows the classic lock-free implementation of Atomic-Increment using CAS [8]. Note that if a thread is delayed at any point while executing this routine, other active threads will be able to proceed with their operations without waiting for the delayed thread, and every time a thread executes a full iteration of the loop, some operation must have made progress. If the CAS succeeds, then the increment of the current thread has taken effect. If the CAS fails, then the value of the counter must have changed during the loop. The only way the counter changes is if a CAS succeeds. Then, some other thread's CAS must have succeeded during the loop and hence that other thread's increment must have taken effect.

2.2 Related Work

The concept of lock-free synchronization goes back more than two decades. It is attributed to early work by Lamport [12] and to the motivating basis for introducing the CAS instruction in the IBM System 370 documentation [8]. The impossibility and universality results of Herlihy [7] had significant influence on the theory and practice of lock-free synchronization, by showing that atomic instructions such as CAS and LL/SC are more powerful than others such as Test-and-Set, Swap, and Fetch-and-Add, in their ability to provide lock-free implementations of arbitrary object types. In other publications [17, 19], we review practical lock-free algorithms for dynamic data structures in light of recent advances in lock-free memory management.

Wilson et. al. [23] present a survey of sequential memory allocation. Berger [2, 3] presents an overview of concurrent allocators, e.g., [4, 6, 10, 11, 13]. In our experiments, we compare our allocator with two widely-used malloc replacement packages for multiprocessor systems, Ptmalloc and Hoard. We also leverage some scalability-enabling high-level structures from Hoard.

Ptmalloc [6], developed by Wolfram Gloger and based on Doug Lea's dlmalloc sequential allocator [14], is part of GNU glibc. It uses multiple arenas in order to reduce the adverse effect of contention. The granularity of locking is the arena. If a thread executing malloc finds an arena locked, it tries the next one. If all arenas are found to be locked, the thread creates a new arena to satisfy its malloc and adds the new arena to the main list of arenas. To improve locality and reduce false sharing, each thread keeps thread-specific information about the arena it used in its last malloc. When a thread frees a chunk (block), it returns the chunk to the arena from which the chunk was originally allocated, and the thread must acquire that arena's lock.

Hoard [2, 3], developed by Emery Berger, uses multiple processor heaps in addition to a global heap. Each heap contains zero or more superblocks. Each superblock contains one or more blocks of the same size. Statistics are maintained individually for each superblock as well as collectively for the superblocks of each heap. When a processor heap is found to have too much available space, one of its superblocks is moved to the global heap. When a thread finds that its processor heap does not have available blocks of the desired size, it checks if any superblocks of the desired size are available in the global heap. Threads use their thread ids to decide which processor heap to use for malloc. For free, a thread must return the block to its original superblock and update the fullness statistics for the superblock as well as the heap that owns it. Typically, malloc and free require one and two lock acquisitions, respectively.

Dice and Garthwaite [5] propose a partly lock-free allocator. The allocator requires special operating system support, which makes it not readily portable across operating systems and programming environments. In the environment for their allocator, the kernel monitors thread migration and preemption and posts upcalls to user-mode. When a thread is scheduled to run, the kernel posts the CPU id of the processor that the thread is to run on during its upcoming time slice. The kernel also saves the user-mode instruction pointer in a thread-specific location and replaces it with the address of a special notification routine that will be the first thing the thread executes when it resumes. The notification routine checks if the thread was in a critical section when it was preempted. If so, the notification routine passes control to the beginning of the critical section instead of the original instruction pointer, so that the thread can retry its critical section. The allocator can apply this mechanism only to CPU-specific data. So, it is only used for the CPU's local heap. For all other operations, such as freeing a block that belongs to a remote CPU heap or any access to the global heap, mutual exclusion locks are used. The allocator is not completely lock-free, and hence—without additional special support from the kernel—it is susceptible to deadlock under arbitrary thread termination or priority inversion.

3. LOCK-FREE ALLOCATOR

This section describes our lock-free allocator in detail. Without loss of generality we focus on the case of a 64-bit address space. The 32-bit case is simpler, as 64-bit CAS is supported on 32-bit architectures.

3.1 Overview

First, we start with the general structure of the allocator. Large blocks are allocated directly from the OS and freed directly to the OS. For smaller block sizes, the heap is composed of large superblocks (e.g., 16 KB). Each superblock is divided into multiple equal-sized blocks. Superblocks are distributed among size classes based on their block sizes. Each size class contains multiple processor heaps proportional to the number of processors in the system. A processor heap contains at most one active superblock. An active superblock contains one or more blocks available for reservation that are guaranteed to be available to threads that reach them through the header of the processor heap. Each superblock is associated with a descriptor. Each allocated block contains a prefix (8 bytes) that points to the descriptor of its superblock. On the first call to malloc, the static structures for the size classes and processor heaps (about 16 KB for a 16 processor machine) are allocated and initialized in a lock-free manner.

Malloc starts by identifying the appropriate processor heap, based on the requested block size and the identity of the calling thread. Typically, the heap already has an active superblock with blocks available for reservation. The thread atomically reads a pointer to the descriptor of the active superblock and reserves a block. Next, the thread atomically

```
// Superblock descriptor structure
typedef anchor :  // fits in one atomic block
  unsigned avail:10,count:10,state:2,tag:42;
  // state codes ACTIVE=0 FULL=1 PARTIAL=2 EMPTY=3

typedef descriptor :
  anchor Anchor;
  descriptor* Next;
  void* sb; // pointer to superblock
  procheap* heap; // pointer to owner procheap
  unsigned sz; // block size
  unsigned maxcount; // superblock size/sz
```

```
// Processor heap structure
typedef active :  unsigned ptr:58,credits:6;
typedef procheap :
  active Active; // initially NULL
  descriptor* Partial; // initially NULL
  sizeclass* sc; // pointer to parent sizeclass

// Size class structure
typedef sizeclass :
  descList Partial; // initially empty
  unsigned sz; // block size
  unsigned sbsize; // superblock size
```

Figure 3: Structures.

pops a block from that superblock and updates its descriptor. A typical free pushes the freed block into the list of available blocks of its original superblock by atomically updating its descriptor. We discuss the less frequent more complicated cases below when describing the algorithms in detail.

3.2 Structures and Algorithms

For the most part, we provide detailed (C-like) code for the algorithms, as we believe that it is essential for understanding lock-free algorithms, unlike lock-based algorithms where sequential components protected by locks can be described clearly using high-level pseudocode.

3.2.1 Structures

Figure 3 shows the details of the above mentioned structures. The `Anchor` field in the superblock descriptor structure contains subfields that can be updated together atomically using CAS or LL/SC. The subfield `avail` holds the index of the first available block in the superblock, `count` holds the number of unreserved blocks in the superblock, `state` holds the state of the superblock, and `tag` is used to prevent the ABA problem as discussed below.

The `Active` field in the processor heap structure is primarily a pointer to the descriptor of the active superblock owned by the processor heap. If the value of `Active` is not `NULL`, it is guaranteed that the active superblock has at least one block available for reservation. Since the addresses of superblock descriptors can be guaranteed to be aligned to some power of 2 (e.g., 64), as an optimization, we can carve a `credits` subfield to hold the number of blocks available for reservation in the active superblock less one. That is, if the value of `credits` is n, then the active superblock contains $n+1$ blocks available for reservation through the `Active` field. Note that the number of blocks in a superblock is not limited to the maximum reservations that can be held in the `credits` subfield. In a typical malloc operation (i.e., when `Active` $\neq$ `NULL` and `credits` > 0), the thread reads `Active` and then atomically decrements `credits` while validating that the active superblock is still valid.

3.2.2 Superblock States

A superblock can be in one of four states: `ACTIVE`, `FULL`, `PARTIAL`, or `EMPTY`. A superblock is `ACTIVE` if it is the active superblock in a heap, or if a thread intends to try to install it as such. A superblock is `FULL` if all its blocks are either allocated or reserved. A superblock is `PARTIAL` if it is not `ACTIVE` and contains unreserved available blocks. A superblock is `EMPTY` if all its blocks are free and it is not `ACTIVE`. An `EMPTY` superblock is safe to be returned to the OS if desired.

3.2.3 Malloc

Figure 4 shows the `malloc` algorithm. The outline of the algorithm is as follows. If the block size is large, then the block is allocated directly from the OS and its prefix is set to indicate the block's size. Otherwise, the appropriate heap is identified using the requested block size and the id of the requesting thread. Then, the thread tries the following in order until it allocates a block: (1) Allocate a block from the heap's active superblock. (2) If no active superblock is found, try to allocate a block from a `PARTIAL` superblock. (3) If none are found, allocate a new superblock and try to install it as the active superblock.

Malloc from Active Superblock

The vast majority of malloc requests are satisfied from the heap's active superblock as shown in the `MallocFromActive` routine in Figure 4. The routine consists of two main steps. The first step (lines 1–6) involves reading a pointer to the active superblock and then atomically decrementing the number of available credits—thereby reserving a block—while validating that the active superblock is still valid. Upon the success of CAS in line 6, the thread is guaranteed that a block in the active superblock is reserved for it.

The second step of `MallocFromActive` (lines 7–18) is primarily a lock-free pop from a LIFO list [8]. The thread reads the index of the first block in the list from `Anchor.avail` in line 8, then it reads the index of the next block in line 10,[1] and finally in line 18 it tries to swing the head pointer (i.e., `Anchor.avail`) atomically to the next block, while validating that at that time what it "thinks" to be the first two indexes in the list (i.e., `oldanchor.avail` and `next`) are indeed the first two indexes in the list, and hence in effect popping the first available block from the list.

Validating that the CAS in line 18 succeeds only if `Anchor.avail` is equal to `oldanchor.avail` follows directly from the semantics of CAS. However, validating that at that time `*addr=next` is more subtle and without the `tag` subfield is susceptible to the ABA problem [8, 19]. Consider the case where in line 8 thread X reads the value A from `Anchor.avail` and in line 10 reads the value B from `*addr`. After line 10, X is delayed and some other thread or threads pop (i.e., allocate) block A then block B and then push (i.e., free) some block C and then block A back in the list. Later, X resumes and executes the CAS in line 18. Without the tag subfield (for simplicity ignore the `count` subfield), the CAS would find `Anchor` equal to `oldanchor` and succeeds where

[1]This is correct even if there is no next block, because in such a case no subsequent malloc will target this superblock before one of its blocks is freed.

```
void* malloc(sz) {
    // Use sz and thread id to find heap.
1   heap = find_heap(sz);
2   if (!heap) // Large block
3     Allocate block from OS and return its address.
    while(1) {
4     addr = MallocFromActive(heap);
5     if (addr) return addr;
6     addr = MallocFromPartial(heap);
7     if (addr) return addr;
8     addr = MallocFromNewSB(heap);
9     if (addr) return addr;
}   }

void* MallocFromActive(heap) {
    do {  // First step:  reserve block
1     newactive = oldactive = heap->Active;
2     if (!oldactive) return NULL;
3     if (oldactive.credits == 0)
4       newactive = NULL;
      else
5       newactive.credits--;
6   } until CAS(&heap->Active,oldactive,newactive);
    // Second step:  pop block
7   desc = mask_credits(oldactive);
    do {
      // state may be ACTIVE, PARTIAL or FULL
8     newanchor = oldanchor = desc->Anchor;
9     addr = desc->sb+oldanchor.avail*desc->sz;
10    next = *(unsigned*)addr;
11    newanchor.avail = next;
12    newanchor.tag++;
13    if (oldactive.credits == 0) {
        // state must be ACTIVE
14      if (oldanchor.count == 0)
15        newanchor.state = FULL;
        else {
16        morecredits =
            min(oldanchor.count,MAXCREDITS);
17        newanchor.count -= morecredits;
        }
      }
18  } until CAS(&desc->Anchor,oldanchor,newanchor);
19  if (oldactive.credits==0 && oldanchor.count>0)
20    UpdateActive(heap,desc,morecredits);
21  *addr = desc; return addr+EIGHTBYTES;
}

UpdateActive(heap,desc,morecredits) {
1   newactive = desc;
2   newactive.credits = morecredits-1;
3   if CAS(&heap->Active,NULL,newactive) return;
    // Someone installed another active sb
    // Return credits to sb and make it partial
    do {
4     newanchor = oldanchor = desc->Anchor;
5     newanchor.count += morecredits;
6     newanchor.state = PARTIAL;
7   } until CAS(&desc->Anchor,oldanchor,newanchor);
8   HeapPutPartial(desc);
}

void* MallocFromPartial(heap) {
retry:
1   desc = HeapGetPartial(heap);
2   if (!desc) return NULL;
3   desc->heap = heap;
    do { // reserve blocks
4     newanchor = oldanchor = desc->Anchor;
5     if (oldanchor.state == EMPTY) {
6       DescRetire(desc); goto retry;
      }
      // oldanchor state must be PARTIAL
      // oldanchor count must be > 0
7     morecredits =
        min(oldanchor.count-1,MAXCREDITS);
8     newanchor.count -= morecredits+1;
9     newanchor.state =
        (morecredits > 0) ?  ACTIVE : FULL;
10  } until CAS(&desc->Anchor,oldanchor,newanchor);
    do { // pop reserved block
11    newanchor = oldanchor = desc->Anchor;
12    addr = desc->sb+oldanchor.avail*desc->sz;
13    newanchor.avail = *(unsigned*)addr;
14    newanchor.tag++;
15  } until CAS(&desc->Anchor,oldanchor,newanchor);
16  if (morecredits > 0)
17    UpdateActive(heap,desc,morecredits);
18  *addr = desc; return addr+EIGHTBYTES;
}

descriptor* HeapGetPartial(heap) {
    do {
1     desc = heap->Partial;
2     if (desc == NULL)
3       return ListGetPartial(heap->sc);
4   } until CAS(&heap->Partial,desc,NULL);
5   return desc;
}

void* MallocFromNewSB(heap) {
1   desc = DescAlloc();
2   desc->sb = AllocNewSB(heap->sc->sbsize);
3   Organize blocks in a linked list starting with index 0.
4   desc->heap = heap;
5   desc->Anchor.avail = 1;
6   desc->sz = heap->sc->sz;
7   desc->maxcount = heap->sc->sbsize/desc->sz;
8   newactive = desc;
9   newactive.credits =
      min(desc->maxcount-1,MAXCREDITS)-1;
10  desc->Anchor.count =
      (desc->maxcount-1)-(newactive.credits+1);
11  desc->Anchor.state = ACTIVE;
12  memory fence.
13  if CAS((&heap->Active,NULL,newactive) {
14    addr = desc->sb;
15    *addr = desc; return addr+EIGHTBYTES;
    } else {
16    Free the superblock desc->sb.
17    DescRetire(desc); return NULL;
    }
}
```

Figure 4: Malloc.

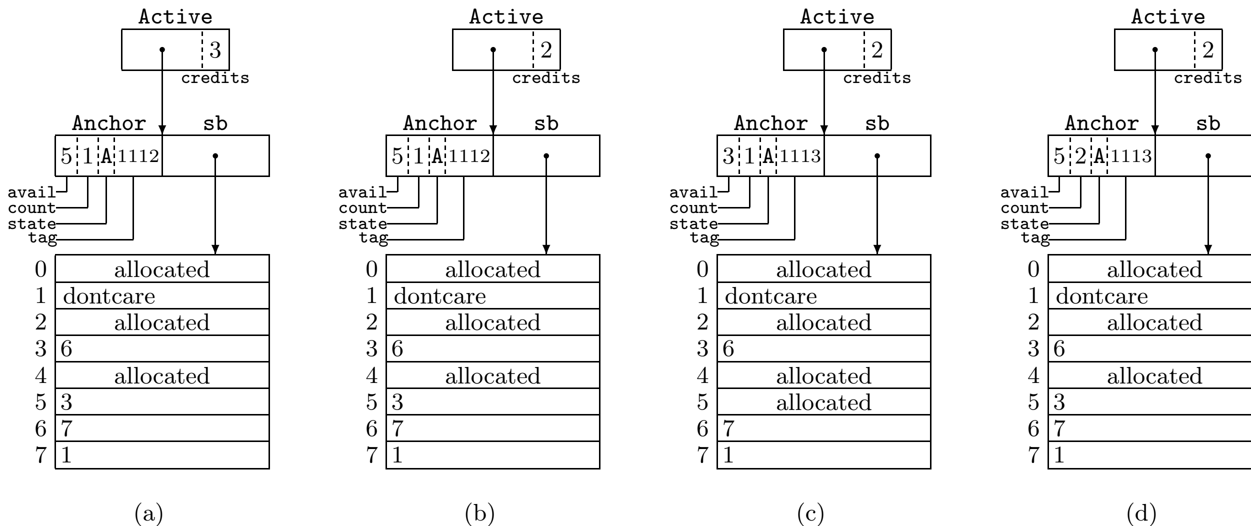


Figure 5: An example of a typical malloc and free from an active superblock. In configuration (a), the active superblock contains 5 available blocks organized in a linked list [5,3,6,7,1], four of which are available for reservation as indicated by `Active.credits`=3. In the first step of malloc, a block is reserved by decrementing `Active.credits`, resulting in configuration (b). In the second step of malloc, block 5 is popped, resulting in configuration (c). Free pushes the freed block (block 5) resulting in configuration (d).

it should not, as the new head of the free list would become block B which is actually not free. Without the tag subfield, X is unable to detect that the value of `Anchor.avail` changed from A to B and finally back to A again (hence the name ABA). To prevent this problem for the `Anchor` field, we use the classic IBM tag mechanism [8]. We increment the `tag` subfield (line 12) on every pop and validate it atomically with the other subfields of `Anchor`. Therefore, in the above mentioned scenario, when the tag is used, the CAS fails—as it should—and X starts over from line 8. The tag must have enough bits to make full wraparound practically impossible in a short time. For an absolute solution for the ABA problem, an efficient implementation of ideal LL/SC—which inherently prevents the ABA problem—using pointer-sized CAS can be used [18, 19].

In lines 13–17, the thread checks if it has taken the last credit in `Active.credit`. If so, it checks if the superblock has more available blocks, either because `maxcount` is larger than `MAXCREDITS` or because blocks were freed. If more blocks are available, the thread reserves as many as possible (lines 16 and 17). Otherwise, it declares the superblock `FULL` (line 15). The reason for doing that is that `FULL` superblocks are not pointed to by any allocator structures, so the first thread to free a block back to a `FULL` superblock needs to know that, in order to take responsibility for linking it back to the allocator structures.

If the thread has taken credits, it tries to update `Active` by executing `UpdateActive`. There is no risk of more than one thread trying to take credits from the same superblock at the same time. Only the thread that sets `Active` to `NULL` in line 6 can do that. Other concurrent threads find `Active` either with `credits`>0 or not pointing to `desc` at all.

Finally the thread stores `desc` (i.e., the address of the descriptor of the superblock from which the block was allocated) into the prefix of the newly allocated block (line 21), so that when the block is subsequently freed, `free` can determine from which superblock it was originally allocated. Each block includes an 8 byte prefix (overhead).

Note that, after a thread finds `Active.credits`>0 and after the success of the CAS in line 6 and before the thread proceeds to a successful CAS in line 18, it is possible that the "active" superblock might have become `FULL` if all available blocks were reserved, `PARTIAL`, or even the `ACTIVE` superblock of a different processor heap (but must be the same size class). However, it cannot be `EMPTY`. These possibilities do not matter to the original thread. After the success of the CAS in line 6, the thread is guaranteed a block from this specific superblock, and all it need do is pop a block from the superblock and leave the superblock's `Anchor.state` unchanged. Figure 5 shows a typical malloc and free from an active superblock.

Updating Active Credits

Typically, when the routine `UpdateActive` in Figure 4 is called, it ends with the success of the CAS operation in line 3 that reinstalls `desc->sb` as the active superblock for `heap` with one or more credits. However, it is possible that after the current thread had set `heap->Active` to `NULL` (line 6 of `MallocFromActive`), some other thread installed a new superblock. If so, the current thread must return the credits, indicate that the superblock is `PARTIAL`, and make the superblock available for future use in line 8 by calling `HeapPutPartial` (described below).

Malloc from Partial Superblock

The thread calls `MallocFromPartial` in Figure 4 if it finds `Active`=`NULL`. The thread tries to get a `PARTIAL` superblock by calling `HeapGetPartial`. If it succeeds, it tries to reserve as many blocks—including one for itself—from the superblock's descriptor. Upon the success of CAS in line 10, the thread is guaranteed to have reserved one or more blocks. It then proceeds in lines 11–15 to pop its reserved block, and if it has reserved more, it deposits the additional credits in `Active` by calling `UpdateActive`.

In `HeapGetPartial`, the thread first tries to pop a superblock from the `Partial` slot associated with the thread's

processor heap. If Partial=NULL, then the thread checks the Partial list associated with the size class as described in Section 3.2.6.

Malloc from New Superblock

If the thread does not find any PARTIAL superblocks, it calls MallocFromNewSB in Figure 4. The thread allocates a descriptor by calling DescAlloc (line 1), allocates a new superblock, and sets its fields appropriately (lines 2–11). Finally, it tries to install it as the active superblock in Active using CAS in line 13. If the CAS fails, the thread deallocates the superblock and retires the descriptor (or alternatively, the thread can take the block, return the credits to the superblock, and install the superblock as PARTIAL). The failure of CAS in line 13 implies that heap->Active is no longer NULL, and therefore a new active superblock must have been installed by another thread. In order to avoid having too many PARTIAL superblocks and hence cause unnecessary external fragmentation, we prefer to deallocate the superblock rather than take a block from it and keep it as PARTIAL.

On systems with memory consistency models [1] weaker than sequential consistency, where the processors might execute and observe memory accesses out of order, *fence* instructions are needed to enforce the ordering of memory accesses. The memory fence instruction in line 12 serves to ensure that the new values of the descriptor fields are observed by other processors before the CAS in line 13 can be observed. Otherwise, if the CAS succeeds, then threads running on other processors may read stale values from the descriptor.[2]

3.2.4 Free

Figure 6 shows the free algorithm. Large blocks are returned directly to the OS. The free algorithm for small blocks is simple. It primarily involves pushing the freed block into its superblock's available list and adjusting the superblock's state appropriately.

The instruction fence in line 14 is needed to ensure that the read in line 13 is executed before the success of the CAS in line 18. The memory fence in line 17 is needed to ensure that the write in line 8 is observed by other processors no later than the CAS in line 18 is observed.

If a thread is the first to return a block to a FULL superblock, then it takes responsibility for making it PARTIAL by calling HeapPutParial, where it atomically swaps the superblock with the prior value in the Partial slot of the heap that last owned the superblock. If the previous value of heap->Partial is not NULL, i.e., it held a partial superblock, then the thread puts that superblock in the partial list of the size class as described in Section 3.2.6.

If a thread frees the last allocated block in a superblock, then it takes responsibility for indicating that the superblock is EMPTY and frees it. The thread then tries to retire the associated descriptor. If the descriptor is in the Partial slot of a processor heap, a simple CAS will suffice to remove it. Otherwise, the descriptor may be in the Partial list of the size class (possibly in the middle). We discuss this case in Section 3.2.6.

[2]Due to the variety in memory consistency models and fence instructions among architectures, it is customary for concurrent algorithms presented in the literature to ignore them. In this paper, we opt to include fence instructions in the code, but for clarity we assume a typical PowerPC-like architecture. However, different architectures—including future ones—may use different consistency models.

```
free(ptr) {
1   if (!ptr) return;
2   ((void**)ptr)--; // get prefix
3   desc = *(descriptor**)ptr;
4   if (large_block_bit_set(desc))
      // Large block - desc holds sz+1
5     { Return block to OS. return; }
6   sb = desc->sb;
    do {
7     newanchor = oldanchor = desc->Anchor;
8     *(unsigned*)ptr = oldanchor.avail;
9     newanchor.avail = (ptr-sb)/desc->sz;
10    if (oldanchor.state == FULL)
11      newanchor.state = PARTIAL;
12    if (oldanchor.count==desc->maxcount-1) {
13      heap = desc->heap;
14      instruction fence.
15      newanchor.state = EMPTY;
      } else
16      newanchor.count++;
17    memory fence.
18  } until CAS(&desc->Anchor,oldanchor,newanchor);
19  if (newanchor.state == EMPTY) {
20    Free the superblock sb.
21    RemoveEmptyDesc(heap,desc);
22  } elseif (oldanchor.state == FULL)
23    HeapPutPartial(desc);
}

HeapPutPartial(desc) {
1   do { prev = desc->heap->Partial;
2      } until CAS(&desc->heap->Partial,prev,desc);
3   if (prev) ListPutPartial(prev);
}

RemoveEmptyDesc(heap,desc) {
1   if CAS(&heap->Partial,desc,NULL)
2     DescRetire(desc);
3   else ListRemoveEmptyDesc(heap->sc);
}
```

Figure 6: Free.

3.2.5 Descriptor List

Figure 7 shows the DescAlloc and DescRetire routines. In DescAlloc, the thread first tries to pop a descriptor from the list of available descriptors (lines 3–4). If not found, the thread allocates a superblock of descriptors, takes one descriptor, and tries to install the rest in the global available descriptor list. In order to avoid unnecessarily allocating too many descriptors, if the thread finds that some other thread has already made some descriptors available (i.e., the CAS in line 8 fails), then it returns the superblock to the OS and starts over in line 1, with the hope of finding an available descriptor. DescRetire is a straightforward lock-free push that follows the classic freelist push algorithm [8].

As mentioned above in the case of the pop operation in the MallocFromActive routine, care must be taken that CAS does not succeed where it should not due to the ABA problem. We indicate this in line 4, by using the term SafeCAS (i.e., ABA-safe). We use the hazard pointer methodology [17, 19]—which uses only pointer-sized instructions—in order to prevent the ABA problem for this structure.

```
descriptor* DescAvail; // initially NULL

descriptor* DescAlloc() {
    while (1) {
1     desc = DescAvail;
2     if (desc) {
3       next = desc->Next;
4       if SafeCAS(&DescAvail,desc,next) break;
      } else {
5       desc = AllocNewSB(DESCSBSIZE);
6       Organize descriptors in a linked list.
7       memory fence.
8       if CAS(&DescAvail,NULL,desc->Next)) break;
9       Free the superblock desc.
      }
    }
10  return desc;
}

DescRetire(desc) {
    do {
1     oldhead = DescAvail;
2     desc->Next = oldhead;
3     memory fence.
4   } until CAS(&DescAvail,oldhead,desc);
}
```

Figure 7: Descriptor allocation.

In the current implementation, superblock descriptors are not reused as regular blocks and cannot be returned to the OS. This is acceptable as descriptors constitute on average less than 1% of allocated memory. However, if desired, space for descriptors can be reused arbitrarily or returned to the OS, by organizing descriptors in a similar manner to regular blocks and maintaining special descriptors for superblocks of descriptor, with virtually no effect on average performance whether contention-free or under high contention. This can be applied on as many levels as desired, such that at most 1% of 1%—and so on—of allocated space is restricted from being reused arbitrarily or returned to the OS.

Similarly, in order to reduce the frequency of calls to `mmap` and `munmap`, we allocate superblocks (e.g., 16 KB) in batches of (e.g., 1 MB) *hyperblocks* (superblocks of superblocks) and maintain descriptors for such hyperblocks, allowing them eventually to be returned to the OS. We organize the descriptor `Anchor` field in a slightly different manner, such that superblocks are not written until they are actually used, thus saving disk swap space for unused superblocks.

3.2.6 Lists of Partial Superblocks

For managing the list of partial superblocks associated with each size class, we need to provide three functions: `ListGetPartial`, `ListPutPartial`, and `ListRemoveEmpty-Desc`. The goal of the latter is to ensure that empty descriptors are eventually made available for reuse, and not necessarily to remove a specific empty descriptor immediately.

In one possible implementation, the list is managed in a LIFO manner, with the possibility of removing descriptors from the middle of the list. The simpler version in [19] of the lock-free linked list algorithm in [16] can be used to manage such a list. `ListPutPartial` inserts `desc` at the head of the list. `ListGetPartial` pops a descriptor from the head of the list. `ListRemoveEmptyDesc` traverses the list until it removes some empty descriptor or reaches the end of the list.

Another implementation, which we prefer, manages the list in a FIFO manner and thus reduces the chances of contention and false sharing. `ListPutPartial` enqueues `desc` at the tail of the list. `ListGetPartial` dequeues a descriptor from the head of the list. `ListRemoveEmptyDesc` keeps dequeuing descriptors from the head of the list until it dequeues a non-empty descriptor or reaches the end of the list. If the function dequeues a non-empty descriptor, then it reenqueues the descriptor at the tail of the list. By removing any one empty descriptor or moving two non-empty descriptor from the head of the list to its end, we are guaranteed that no more than half the descriptors in the list are left empty. We use a version of the lock-free FIFO queue algorithm in [20] with optimized memory management for the purposes of the new allocator.

For preventing the ABA problem for pointer-sized variables in the above mentioned list implementations, we cannot use IBM ABA-prevention tags (such as in `Anchor.tag`), instead we use ideal LL/SC constructions using pointer-sized CAS [18]. Note that these constructions as described in [18] use memory allocation, however a general-purpose malloc is not needed. In our implementation we allocate such blocks in a manner similar but simpler than allocating descriptors.

Note that in our allocator, unlike Hoard [3], we do not maintain fullness classes or keep statistics about the fullness of processor heaps and we are quicker to move partial superblocks to the partial list of the size class. This simplicity allows lower latency and lower fragmentation. But, one concern may be that this makes it more likely for blocks to be freed to a superblock in the size class partial lists. However, this is not a disadvantage at all, unlike Hoard [3] and also [5] where this can cause contention on the *global* heap's lock. In our allocator, freeing a block into such a superblock does not cause any contention with operations on other superblocks, and in general is no more complex or less efficient than freeing a block into a superblock that is in the thread's own processor heap. Another possible concern is that by moving partial superblocks out of the processor heap *too quickly*, contention and false sharing may arise. This is why we use a most-recently-used Partial slot (multiple slots can be used if desired) in the processor heap structure, and use a FIFO structure for the size class partial lists.

4. EXPERIMENTAL RESULTS

In this section, we describe our experimental performance results on two PowerPC multiprocessor systems. The first system has sixteen 375 MHz POWER3-II processors, with 24 GB of memory, 4 MB second level caches. The second system has eight 1.2 GHz POWER4+ processors, with 16 GB of memory, 32 MB third level caches. Both systems run AIX 5.1. We ran experiments on both systems. The results on the POWER3 system (with more processors) provided more insights into the allocators' scalability and ability to avoid false sharing and contention. The results on the POWER4 system provided insights into the contention-free latency of the allocators and contention-free synchronization costs on recent processor architectures.

We compare our allocator with the default AIX libc malloc,[3] Hoard [3] version 3.0.2 (December 2003), and Ptmal-

[3]Our observations on the default libc malloc are based on external experimentation only and are not based on any knowledge of its internal design.

	375 MHz POWER3-II			1.2 GHz POWER4+		
	New	Hoard	Ptmalloc	New	Hoard	Ptmalloc
Linux scalability	2.25	1.11	1.83	2.75	1.38	1.92
Threadtest	2.18	1.20	1.94	2.35	1.23	1.97
Larson	2.90	2.22	2.53	2.95	2.37	2.67

Table 1: Contention-free speedup over libc malloc.

loc2 (Nov. 2002) [6]. All allocators and benchmarks were compiled using gcc and g++ with the highest optimization level (-O6) in 64-bit mode. We used pthreads for multithreading. All allocators were dynamically linked as shared libraries. For meaningful comparison, we tried to use optimal versions of Hoard and Ptmalloc as best we could. We modified the PowerPC lightweight locking routines in Hoard by removing a `sync` instruction from the beginning of the lock acquisition path, replacing the `sync` at the end of lock acquisition with `isync`, and adding `eieio` before lock release. These changes reduced the average contention-free latency of a pair of malloc and free using Hoard from 1.76 μs. to 1.51 μs. on POWER3, and from 885 ns. to 560 ns. on POWER4. The default distribution of Ptmalloc2 uses pthread mutex for locking. We replaced calls to pthread mutex by calls to a lightweight mutex that we coded using inline assembly. This reduced the average contention-free latency of a pair of malloc and free using Ptmalloc by more than 50%, from 1.93 μs. to 923 ns. on POWER3 and from 812 ns. to 404 ns. on POWER4. In addition, Ptmalloc showed substantially better scalability using the lightweight locks than it did using pthread mutex locks.

4.1 Benchmarks

Due to the lack of standard benchmarks for multithreaded dynamic memory allocation, we use microbenchmarks that focus on specific performance characteristics. We use six benchmarks: benchmark 1 of *Linux Scalability* [15], *Threadtest*, *Active-false*, and *Passive-false* from Hoard [3], *Larson* [13], and a lock-free producer-consumer benchmark that we describe below.

In *Linux scalability*, each thread performs 10 million malloc/free pairs of 8 byte blocks in a tight loop. In *Threadtest*, each thread performs 100 iterations of allocating 100,000 8-byte blocks and then freeing them in order. These two benchmarks capture allocator latency and scalability under regular private allocation patterns.

In *Active-false*, each thread performs 10,000 malloc/free pairs (of 8 byte blocks) and each time it writes 1,000 times to each byte of the allocated block. *Passive-false* is similar to *Active-false*, except that initially one thread allocates blocks and hands them to the other threads, which free them immediately and then proceed as in *Active-false*. These two benchmarks capture the allocator's ability to avoid causing false sharing [22] whether actively or passively.

In *Larson*, initially one thread allocates and frees random sized blocks (16 to 80 bytes) in random order, then an equal number of blocks (1024) is handed over to each of the remaining threads. In the parallel phase which lasts 30 seconds, each thread randomly selects a block and frees it, then allocates a new random-sized block in its place. The benchmark measures how many free/malloc pairs are performed during the parallel phase. *Larson* captures the robustness of malloc's latency and scalability under irregular allocation patterns with respect to block-size and order of deallocation over a long period of time.

In the lock-free *Producer-consumer* benchmark, we measure the number of tasks performed by t threads in 30 seconds. Initially, a database of 1 million items is initialized randomly. One thread is the producer and the others, if any, are consumers. For each task, the producer selects a random-sized (10 to 20) random set of array indexes, allocates a block of matching size (40 to 80 bytes) to record the array indexes, then allocates a fixed size task structure (32 bytes) and a fixed size queue node (16 bytes), and enqueues the task in a lock-free FIFO queue [19, 20]. A consumer thread repeatedly dequeues a task, creates histograms from the database for the indexes in the task, and then spends time proportional to a parameter *work* performing local work similar to the work in Hoard's *Threadtest* benchmark. When the number of tasks in the queue exceeds 1000, the producer helps the consumers by dequeuing a task from the queue and processing it. Each task involves 3 malloc operations on the part of the producer, and one malloc and 4 free operations on the part of the consumer. The consumer spends substantially more time on each task that the producer. *Producer-consumer* captures malloc's robustness under the producer-consumer sharing pattern, where threads free blocks allocated by other threads.

4.2 Results

4.2.1 Latency

Table 1 presents contention-free[4] speedups over libc malloc for the new allocator, Hoard, and Ptmalloc, for the benchmarks that are affected by malloc latency: *Linux scalability*, *Threadtest*, and *Larson*. Malloc's latency had little or no effect on the performance of *Active-false*, *Passive-false*, and *Producer-consumer*.

The new allocator achieves significantly lower contention-free latency than the other allocators under both regular and irregular allocation patterns. The reason is that it has a faster execution path in the common case. Also, unlike lock-based allocators, it operates only on the actual allocator variables without the need to operate on additional lock related variables and to synchronize these accesses with the accesses to the allocator variables through fence instructions.

The new allocator requires only one memory fence instruction (line 17 of `free`) in the common case for each pair of malloc and free, while every lock acquisition and release requires an instruction fence before the critical section to pre-

[4]It appears that libc malloc as well as Hoard use a technique where the parent thread bypasses synchronization if it knows that it has not spawned any threads yet. We applied the same technique to our allocator and the average single-thread latency for our allocator was lower than those for libc malloc and Hoard. However, in order to measure true contention-free latency under multithreading, in our experiments, the parent thread creates an additional thread at initialization time which does nothing and exits immediately before starting time measurement.

vent reads inside the critical section from reading stale data before lock acquisition, and a memory fence after the end of the critical section to ensure that the lock is not observed to be free before the writes inside the critical sections are also observed by other processors. In the common case, a pair of malloc and free using Ptmalloc and Hoard need to acquire and release two and three locks, respectively.

Interestingly, when we conducted experiments with a lightweight test-and-set mutual exclusion lock on the POWER4 system, we found that the average contention-free latency for a pair of lock acquire and release is 165 ns. On the other hand. the average contention-free latency for a pair of malloc and free in *Linux Scalability* using our allocator is 282 ns., i.e., it is less than twice that of a minimal critical section protected by a lightweight test-and-set lock. That is, on that architecture, it is highly unlikely—if not impossible—for a lock-based allocator (without per-thread private heaps) to have lower latency than our lock-free allocator, even if it uses the fastest lightweight lock to protect malloc and free and does nothing in these critical sections.

4.2.2 Scalability and Avoiding False Sharing

Figure 8(a) shows speedup results relative to contention-free libc malloc for *Linux scalability*. Our allocator, Ptmalloc, and Hoard scale well with varying slopes proportional to their contention-free latency. Libc malloc does not scale at all, its speedup drops to 0.4 on two processors and continues to decline with more processors. On 16 processors the execution time of libc malloc is 331 times as much as that of our allocator.

The results for *Threadtest* (Figure 8(b)) show that our allocator and Hoard scale in proportion to their contention-free latencies. Ptmalloc scales but at a lower rate under high contention, as it becomes more likely that threads take over the arenas of other threads when their own arenas have no free blocks available, which increases the chances of contention and false sharing.

Figures 8(c–d) show the results for *Active-false* and *Passive-false*. The latency of malloc itself plays little role in these results. The results reflect only the effect of the allocation policy on inducing or avoiding false sharing. Our allocator and Hoard are less likely to induce false sharing than Ptmalloc and libc malloc.

In *Larson* (Figure 8(e)), which is intended to simulate server workloads, our allocator and Hoard scale, while Ptmalloc does not, probably due to frequent switching of threads between arenas, and consequently more frequent cases of freeing blocks to arenas locked by other threads. We also noticed, when running this benchmark, that Ptmalloc creates more arenas than the number of threads, e.g., 22 arenas for 16 threads, indicating frequent switching among arenas by threads. Even though freeing blocks to remote heaps in Hoard can degrade performance, this effect is eliminated after a short time. Initially threads free blocks that were allocated by another thread, but then in the steady state they free blocks that they have allocated from their own processor heaps.

4.2.3 Robustness under Producer-Consumer

For *Producer-consumer* we ran experiments with various values for *work* (parameter for local work per task). Figures 8(f–h) show the results for *work* set to 500, 750, and 1000, respectively. The results for all the allocators are virtually identical under no contention, thus the latency of the allocator plays a negligible role in the results for this benchmark. The purpose of this benchmark is to show the robustness of the allocators under the producer-consumer sharing pattern when the benchmark is scalable. The case where the benchmark cannot scale even using a perfect allocator is not of interest. We focus on the knee of the curve, where the differences in robustness between allocators impact the scalability of the benchmark.

Our allocator scales perfectly with work set to 1000 and 750, and up to 13 processors with work set to 500. With more than 13 processors (and with work set to 500), we found that the producer could not keep up with the consumers (as the queue was always empty at the end of each experiment), which is not an interesting case as the application would not be scalable in any case. Our allocator's scalability is limited only by the scalability of the application.

Ptmalloc scales to a lesser degree, but at the cost of higher external memory fragmaentation, as the producer keeps creating and switching arenas due to contention with consumers, even though most arenas already have available blocks.

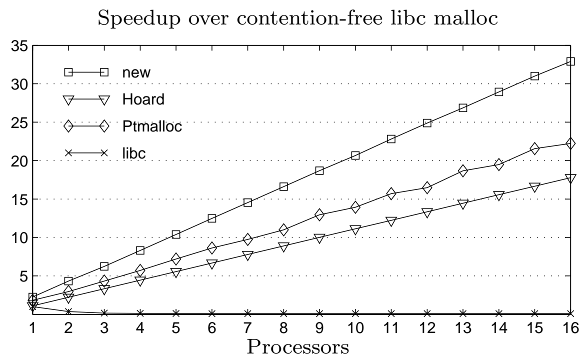
Hoard's scalability suffers due to high contention on the producer's heap, as 75% of all malloc and free operations are targeted at the same heap. Our allocator's performance does not suffer, although it faces exactly the same situation. The main reason is that in Hoard, even in the common case, free operations need to acquire either the processor heap's lock or the global heap's lock. In our allocator typical free operations are very simple and operate only on the superblock descriptor associated with the freed block, thus allowing substantially more concurrency than Hoard. Other minor reasons for our allocator's ability to perform well even under contention on the same superblock are: (a) In our allocator, read-modify-write code segments are shorter in duration, compared with critical sections in Hoard. (b) Successful lock-free operations can overlap in time, while mutual exclusion locks by definition must strictly serialize critical sections.

4.2.4 Optimization for Uniprocessors

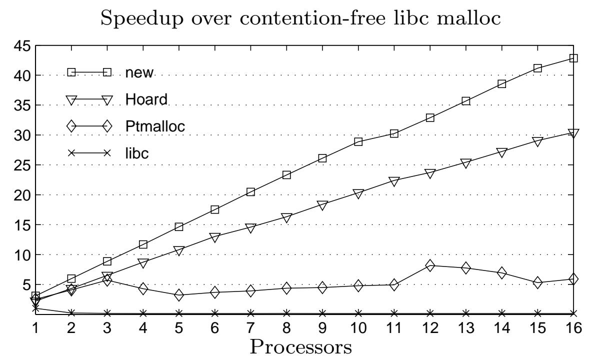
With uniprocessors in mind, we modified a version of our allocator such that threads use only one heap, and thus when executing malloc, threads do not need to know their id. This optimization achieved 15% increase in contention-free speedup on *Linux scalability* on POWER3. When we used multiple threads on the same processor, performance remained unaffected, as our allocator is preemption-tolerant. In practice, the allocator can determine the number of processors in the system at initialization time by querying the system environment.

4.2.5 Space Efficiency

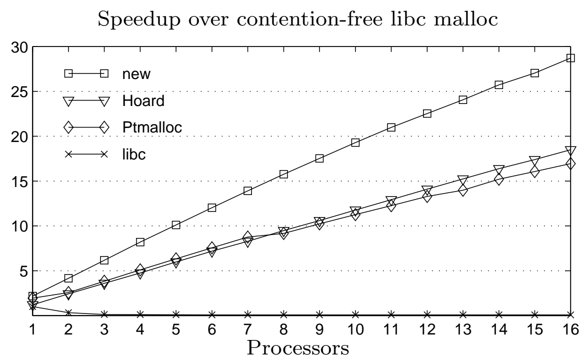
We tracked the maximum space used by our allocator, Hoard, and Ptmalloc when running the benchmarks that allocate a large number of blocks: *Threadtest*, *Larson*, and *Producer-consumer*. The maximum space used by our allocator was consistently slightly less than that used by Hoard, as in our allocator each processor heap holds at most two superblocks, while in Hoard each processor heap holds a variable number of superblocks proportional to allocated blocks. The maximum space allocated by Ptmalloc was consistently more than that allocated by Hoard and our allocator. The ratio of the maximum space allocated by Ptmalloc to the maximum space allocated by ours, on 16 processors, ranged from 1.16 in *Threadtest* to 3.83 in *Larson*.



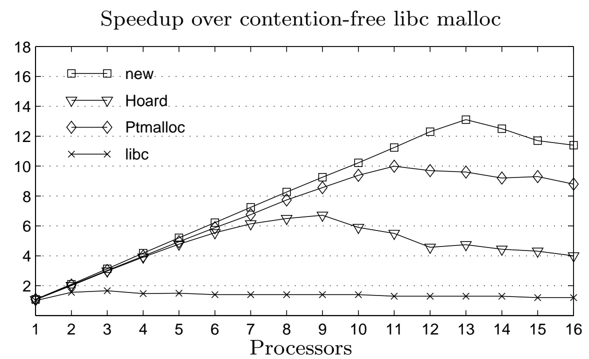
(a) Linux scalability



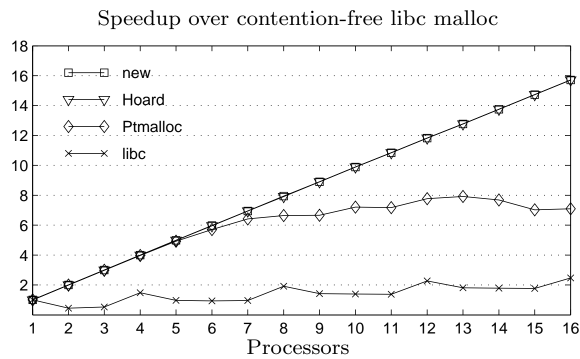
(e) Larson



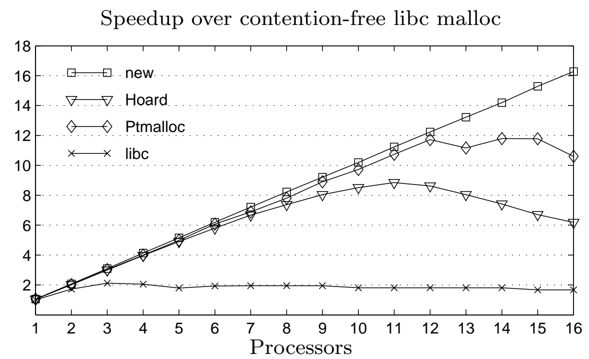
(b) Threadtest



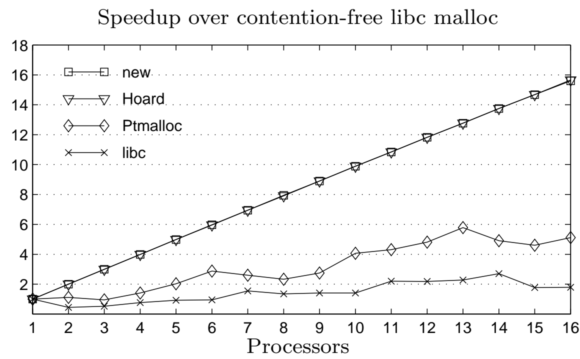
(f) Producer-consumer with work = 500



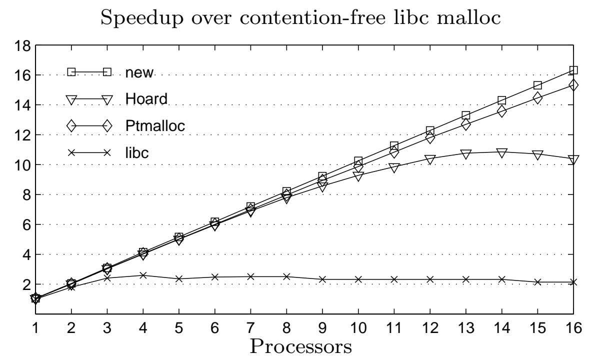
(c) Active false sharing



(g) Producer-consumer with work = 750



(d) Passive false sharing



(h) Producer-consumer with work = 1000

Figure 8: Speedup results on 16-way 375 MHz POWER3.

5. SUMMARY

In this paper we presented a completely lock-free dynamic memory allocator. Being completely lock-free, our allocator is immune to deadlock regardless of scheduling policies and even when threads may be killed arbitrarily. Therefore, it can offer async-signal-safety, tolerance to priority inversion, kill-tolerance, and preemption-tolerance, without requiring any special kernel support or incurring performance overhead. Our allocator is portable across software and hardware platforms, as it requires only widely-available OS support and hardware atomic primitives. It is general-purpose and does not impose any unreasonable restrictions regarding the use or initialization of the address space. It is space efficient and limits space blowup [3] to a constant factor.

Our experimental results compared our allocator with the default AIX 5.1 libc malloc, and two of the best multithread allocators, Hoard [3] and Ptmalloc [6]. Our allocator outperformed the other allocators in all cases, often by significant margins, under various levels of parallelism and allocation patterns. Our allocator showed near perfect scalability under various allocation and sharing patterns. Under maximum contention on 16 processors, it achieved a speedup of 331 over libc malloc.

Equally significant, our allocator offers substantially lower latency than the other allocators. Under no contention, it achieved speedups of 2.75, 1.99, and 1.43 over libc malloc, and highly-optimized versions of Hoard and Ptmalloc, respectively. Scalable allocators are often criticized that they achieve their scalability at the cost of higher latency in the more common case of no contention. Our allocator achieves both scalability and low latency, in addition to many other performance and qualitative advantages.

Furthermore, this work, in combination with recent lock-free methods for safe memory reclamation [17, 19] and ABA prevention [18] that use only single-word CAS, allows lock-free algorithms including efficient algorithms for important object types—such as LIFO stacks [8], FIFO queues [20], and linked lists and hash tables [16, 21]—to be both completely dynamic and completely lock-free, including in 64-bit applications and on systems without support for automatic garbage collection, all efficiently without requiring special OS support and using only widely-available 64-bit atomic instructions.

Acknowledgments

The author thanks Emery Berger, Michael Scott, Yefim Shuf, and the anonymous referees for valuable comments on the paper.

6. REFERENCES

[1] Sarita V. Adve and Kourosh Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Computer*, 29(12):66–76, 1996.

[2] Emery D. Berger. *Memory Management for High-Performance Applications*. PhD thesis, University of Texas at Austin, August 2002.

[3] Emery D. Berger, Kathryn S. McKinley, Robert D. Blumofe, and Paul R. Wilson. Hoard: A scalable memory allocator for multithreaded applications. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 117–128, November 2000.

[4] Bruce M. Bigler, Stephen J. Allan, and Rodney R. Oldehoeft. Parallel dynamic storage allocation. In *Proceedings of the 1985 International Conference on Parallel Processing*, pages 272–275, August 1985.

[5] Dave Dice and Alex Garthwaite. Mostly lock-free malloc. In *Proceedings of the 2002 International Symposium on Memory Management*, pages 269–280, June 2002.

[6] Wolfram Gloger. *Dynamic Memory Allocator Implementations in Linux System Libraries*. `http://www.dent.med.uni-muenchen.de/~wmglo/`.

[7] Maurice P. Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, January 1991.

[8] IBM. *IBM System/370 Extended Architecture, Principles of Operation*, 1983. Publication No. SA22-7085.

[9] IEEE. *IEEE Std 1003.1, 2003 Edition*, 2003.

[10] Arun K. Iyengar. *Dynamic Storage Allocation on a Multiprocessor*. PhD thesis, MIT, 1992.

[11] Arun K. Iyengar. Parallel dynamic storage allocation algorithms. In *Proceedings of the Fifth IEEE Symposium on Parallel and Distributed Processing*, pages 82–91, December 1993.

[12] Leslie Lamport. Concurrent reading and writing. *Communications of the ACM*, 20(11):806–811, November 1977.

[13] Per-Åke Larson and Murali Krishnan. Memory allocation for long-running server applications. In *Proceedings of the 1998 International Symposium on Memory Management*, pages 176–185, October 1998.

[14] Doug Lea. *A Memory Allocator*. `http://gee.cs.oswego.edu/dl/html/malloc.html`.

[15] Chuck Lever and David Boreham. Malloc() performance in a multithreaded Linux environment. In *Proceedings of the FREENIX Track of the 2000 USENIX Annual Technical Conference*, June 2000.

[16] Maged M. Michael. High performance dynamic lock-free hash tables and list-based sets. In *Proceedings of the Fourteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 73–82, August 2002.

[17] Maged M. Michael. Safe memory reclamation for dynamic lock-free objects using atomic reads and writes. In *Proceedings of the Twenty-First Annual ACM Symposium on Principles of Distributed Computing*, pages 21–30, July 2002.

[18] Maged M. Michael. ABA prevention using single-word instructions. Technical Report RC 23089, IBM T. J. Watson Research Center, January 2004.

[19] Maged M. Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Transactions on Parallel and Distributed Systems*, 2004. To appear. See `www.research.ibm.com/people/m/michael/pubs.htm`.

[20] Maged M. Michael and Michael L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing*, pages 267–275, May 1996.

[21] Ori Shalev and Nir Shavit. Split-ordered lists: Lock-free extensible hash tables. In *Proceedings of the Twenty-Second Annual ACM Symposium on Principles of Distributed Computing*, pages 102–111, July 2003.

[22] Josep Torrellas, Monica S. Lam, and John L. Hennessy. False sharing and spatial locality in multiprocessor caches. *IEEE Transactions on Computers*, 43(6):651–663, June 1994.

[23] Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles. Dynamic storage allocation: A survey and critical review. In *Proceedings of the 1995 International Workshop on Memory Management*, pages 1–116, September 1995.