# 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.


*PLDI'04,* June 9–11, 2004, Washington, DC, USA.


## 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