A Modern and Competitive Lock-Free Dynamic Memory Allocator

Ricardo Leite and Ricardo Rocha

CRACS & INESC TEC and Faculty of Sciences, University of Porto
Rua do Campo Alegre, 1021/1055, 4169-007 Porto, Portugal
{rleite,ricroc}@dcc.fc.up.pt

Abstract. This paper presents LRMalloc, a lock-free memory allocator that leverages lessons of modern memory allocators and combines them with a lock-free scheme. Current state-of-the-art memory allocators possess good performance but lack desirable lock-free properties, such as, priority inversion tolerance, kill-tolerance availability, and/or deadlock and livelock immunity. LRMalloc's purpose is to show the feasibility of lock-free memory management algorithms, without sacrificing competitiveness in comparison to commonly used state-of-the-art memory allocators, especially for concurrent multithreaded applications.

Keywords: Memory Allocator · Lock-Freedom · Implementation.

1 Introduction

Dynamic memory allocation is an important component of most applications. Nowadays, due to the use of thread-level parallelism by applications, an important requirement in a memory allocator implementation is the ability to be multithreading safe (or MT-safe). To guarantee MT-safety, current state-of-the-art memory allocators use *lock-based data structures*. The use of locks has implications in terms of program performance, availability, robustness and flexibility. As such, current memory allocators attempt to minimize the effects of locking by using fine-grained locks, or by providing synchronization-free allocations as much as possible through the use of thread-specific caches.

However, even with only infrequent and fine-grained locking, lock-based memory allocators are still subject to a number of disadvantages. They are vulnerable to deadlocks and livelocks, prune to priority inversion and delays due to preemption during locking, and incapable of dealing with unexpected thread termination. A more challenging but desirable alternative to ensure MT-safety is to use *lock-free synchronization*. Lock-freedom guarantees system-wide progress whenever a thread executes some finite amount of steps, whether by the thread itself or by some other thread in the process. By definition, lock-free synchronization uses no locks and does not obtain mutual exclusive access to a resource at any point. It is therefore immune to deadlocks and livelocks. Without locks, priority inversion and delays due to preemption during locking cannot occur, and unexpected thread termination is also not problematic.

More importantly, a truly lock-free data structure or algorithm that needs to dynamically allocate memory cannot be built on top of a lock-based memory allocator without compromising the lock-freedom property. This is particularly relevant for lock-free data structures or algorithms that use memory reclamation techniques such as *epochs* or *hazard pointers* [6].

In this work, we present the design and implementation of LRMalloc, a lock-free dynamic memory allocator that leverages lessons of modern allocators to show the feasibility of lock-free memory management, without sacrificing performance in comparison to commonly used state-of-the-art allocators.

The remainder of the paper is organized as follows. First, we introduce some background and related work. Next, we describe the key data structures and algorithms that support the implementation of LRMalloc. Then, we show experimental results comparing LRMalloc against other modern and relevant memory allocators. We end by outlining conclusions and discussing further work.

2 Background & Related Work

Memory allocation as a research topic has a rich history. Starting with the 1960s, and motivated by the exorbitant cost of main memory, researchers poured effort into the design of allocator strategies and algorithms that minimized *memory fragmentation* – a fundamental problem in memory management that implies needing more memory to satisfy a sequence of memory requests than the sum of the memory required for those memory requests. In this regard, numerous strategies and algorithms were proposed and developed to reduce memory fragmentation. Other fundamental concepts, such as *size classes* (e.g., segregated storage), also appeared in this time period, and are still used in state-of-the-art memory allocators. For an excellent survey, that summarizes most memory allocation research prior to 1995, please see Wilson *et al.* [12].

More recently, with the decrease in cost of main memory and the appearance of computer systems with higher and higher core counts, memory allocation research has taken a different path. Focus has shifted into memory allocator performance and scalability, often in detriment of fragmentation. Problems specific to concurrent memory allocation, such as *false sharing* and *blowup*, have also received attention, being first noted and solved by Berger *et al.* in the Hoard memory allocator [1].

Instead of mutual exclusion, lock-free algorithms use atomic instructions provided by the underlying architecture to guarantee consistency in multithreaded execution. The most relevant atomic instruction is CAS (*Compare-and-Swap*), which is widely supported in just about every modern architecture either directly or by efficient emulation by using *Linked Load* and *Store Conditional* instructions. Listing 1.1 shows the pseudo-code for the CAS instruction.

Lock-free data structures and algorithms that aim to be widely portable should be CAS-based. However CAS-based designs are vulnerable to the ABA problem – a fundamental and challenging problem which complicates the implementation and design of lock-free data structures and algorithms [2].

Listing 1.1. Compare-and-Swap

```
1 bool CAS(int* address, int expected, int desired) {
2   if (*address == expected) {
3     *address = desired;
4     return true;
5   }
6   return false;
7 }
```

The first lock-free segregated fit mechanism was introduced by Michael [11]. Michael's allocator is particularly relevant because it only makes use of the widely available and portable CAS operation for lock-free synchronization. However, as our experimental results will show (in Section 4), compared to modern alternatives, Michael's allocator is slow as it uses several atomic operations per allocation and deallocation, operations which are expensive in today's computer architectures. In order to succeed, lock-free memory allocators must amortize the cost of an atomic operation through several allocation and deallocation requests.

Despite the relatively large number of memory allocators developed over the years, there are only a few that are widely used. The most commonly used memory allocator is by far Ptmalloc2 [5], the standard memory allocator distributed with glibc. It is a modified version of Doug Lea's memory allocator [8], a fragmentation-focused memory allocator that uses a *deferred coalescing scheme*. Ptmalloc2 has been updated over time with features to improve its multithreaded performance, such as the use of *arenas* (several independent allocators that can be used simultaneously thus reducing thread contention) and *thread caches* for small block sizes. Applications that are focused on multithreaded performance usually take note of the performance degradation of the standard C lib allocator and change to a different allocator, such as Jemalloc [3] or TCMalloc [4].

Jemalloc is a state-of-the-art memory allocator that focuses on fragmentation avoidance and scalable concurrency support. It is the default memory allocator on the FreeBSD operating system, and is widely used in numerous multithreaded applications. At process start, it creates a number of *arenas* equal to 4 times the number of CPUs in the system in an effort to minimize lock contention. Threads are assigned to the arena with the lowest number of assigned threads. Like other memory allocators, Jemalloc uses thread caches and defines size classes for allocations, but also uses coalescing strategies in an effort to decrease fragmentation.

TCMalloc is a memory allocator which focuses on thread-caching. Unlike other allocators, it does not use arenas to decrease contention on shared data structures. It instead uses fine-grained locks and implements a number of novel cache management algorithms [9] to improve average-case latency, scalability and decrease cache-provoked fragmentation.

3 LRMalloc

This section describes LRMalloc in more detail. We assume implementation on a 64-bit system architecture (e.g., x86-64) and, for the sake of brevity, we only show C-like code for the key implementation routines[1].

[1] Source code is available in https://github.com/ricleite/lrmalloc.

3.1 High-Level Overview

From a high-level perspective, LRMalloc is divided into three main components: (i) the *thread caches*, one per thread; (ii) the *heap*; and (iii) the *pagemap*. Figure 1 shows the relationship between these three components, the user's application and the operating system.

Similarly to other memory allocators, the thread caches are a synchronization-free component which uses a stack implemented as a singly-linked list to store (a finite number of) blocks that are available for use. A separate stack is kept for each size class. Each thread cache is meant to handle the common allocation case, where a **malloc()** becomes a single stack pop and a **free()** is a stack push using the appropriate size class. When the cache is empty, a new list of blocks is fetched from the heap. When the cache is full, blocks are flushed back to the heap. This simple and speed-efficient common case is essential for a competitive memory allocator. In LRMalloc, size classes are generated according to the following series (series also adopted by Jemalloc [3]):

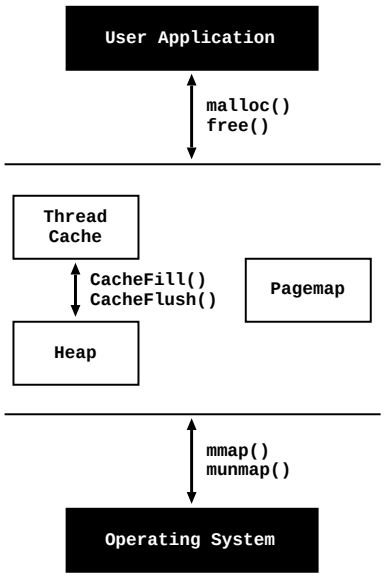


Fig. 1. LRMalloc's overview

1. 2^X
2. $2^X + 2^{(X-2)}$
3. $2^X + 2^{(X-1)}$
4. $2^X + 2^{(X-1)} + 2^{(X-2)}$
5. $2^{(X+1)}$ (repeat, same as first case)

Note that not all values generated by the series are valid due to alignment requirements on the C standard, and thus those are removed. Without those cases, the above series limits internal fragmentation (unused space inside a block given to the application) to a maximum of 25% of the allocated memory.

The heap and the pagemap are lock-free components. The heap manages *superblocks* from which it carves blocks to be used by thread caches. Superblocks are continuous set of pages, which may either be used to carve up blocks of the same size class or for a *single large allocation*. Similarly to other allocators, in LRMalloc, large allocations are allocations larger than the largest size class. The pagemap stores metadata for pages used by superblocks. Its raison d'être is to find out metadata of a given block, such as the size class and superblock the block belongs to. Therefore, the pagemap is merely a bookkeeping component, kept up to date by the heap. The pagemap stores metadata on a per-page basis instead of a per-block basis, which reduces the amount of memory used for bookkeeping and increases the locality of blocks provided to the application.

Listing 1.2 shows the **malloc()** and **free()** high-level routines. Memory allocation starts by computing the size class corresponding to the requested size. In an effort to decrease internal fragmentation, there are only size classes for *small allocations*. LRMalloc's size classes go up to 16KB (4 pages). Allocations that are larger than the largest size class are treated as *large allocations*, and handled differently. For large allocations, LRMalloc creates a superblock with the appropriate size in number of pages through **mmap()**. When a large allocation is **free()**'d, the corresponding superblock is **munmap()** and thus returned to the operating system. Large allocations are identified with a size class equal to 0. If the allocation isn't large, the cache corresponding to the size class is accessed and checked. In the common case, the cache will not be empty and thus a block will be fetched from the cache with a pop operation. In the uncommon case, the cache is empty and thus it must be filled with **CacheFill()**.

Listing 1.2. High-level allocation and deallocation routines

```
1  void* malloc(size_t size) {
2    size_t scIdx = ComputeSizeClass(size);
3    if (scIdx == 0)                                  // large allocation
4      return AllocateLargeBlock();
5    Cache* cache = GetCache(scIdx);
6    if (CacheIsEmpty(cache))
7      CacheFill(scIdx, cache);
8    return CachePopBlock(cache);
9  }
10
11 void free(void* ptr) {
12   size_t scIdx = GetSizeClassFromPageMap(ptr);      // get metadata
13   if (scIdx == 0)                                  // large allocation
14     return DeallocateLargeBlock();
15   Cache* cache = GetCache(scIdx);
16   if (CacheIsFull(cache))
17     CacheFlush(scIdx, cache);
18   CachePushBlock(cache, ptr);
19 }
```

Memory deallocation also starts by finding out the size class of the provided allocation. This is the step where the pagemap component becomes relevant, as it keeps metadata about all allocated pages. As before, large allocations are handled differently, and for small allocations the corresponding cache is accessed. In the common case, the cache will not be full and thus the allocation will just be added to the cache. In the uncommon case, the cache is full and thus **CacheFlush()** must be called to reduce the number of blocks in the cache.

Caches are thread-specific objects, and thus all operations using the cache are synchronization-free. In the allocation and deallocation algorithms, only the **CacheFill()** and **CacheFlush()** routines require synchronization. Both routines are described in more detail next. For the sake of brevity, we will omit the **GetCache()**, **CacheIsEmpty()**, **CacheIsFull()**, **CachePopBlock()** and **CachePushBlock()** subroutines, which are trivially implemented.

3.2 Heap

Due to its lock-free nature, the LRMalloc's heap component is by far the most complex component of the allocator. It is based on Michael's lock-free memory

allocation algorithm [11] but adapted with a number of improvements to work with the presence of thread caches. The heap manages superblocks through *descriptors*. Descriptors are unreclaimable but reusable objects used by the heap to track superblocks. Descriptors contain the superblock's metadata, such as, where the superblock begins, the size class of its blocks, and the number of blocks it contains. It also includes an *anchor*, an inner structure that describes the superblock's state and is small enough to fit inside a single word to be atomically updated with CAS instructions. Listing 1.3 presents the anchor and descriptor struct definitions. The **Anchor.avail** field refers to the first available block in the superblock. A free block then points to the next free block on the chain. The exact number of bits used by the **Anchor.avail** and **Anchor.count** fields (31 in the current implementation) are implementation independent and can be adjusted to how large superblocks can be and to how many blocks at most they can have. In our proposal, and unlike Michael's original algorithm, the anchor does not need a ABA prevention tag.

Listing 1.3. Anchor and descriptor structures

```
1  struct Anchor {
2    size_t state : 2;   // may be EMPTY = 0, PARTIAL = 1 or FULL = 2
3    size_t avail : 31;                  // index of first free block
4    size_t count : 31;                      // number of free blocks
5  };
6
7  struct Descriptor {
8    Anchor anchor;
9    char* superblock;                         // pointer to superblock
10   size_t blocksize;            // size of each block in superblock
11   size_t maxcount;                               // number of blocks
12   size_t scIdx;              // size class of blocks in superblock
13 };
```

Listing 1.4 describes the **CacheFill()** algorithm. By default, the algorithm tries to reuse a partially free superblock by calling **CacheFillFromPartialSB()**, which corresponds to start by trying to get a descriptor from a lock-free stack (**HeapGetPartialSB()** in line 10). If there is such a descriptor, it may point to a superblock where all blocks have been freed, in which case the superblock has been returned to the operating system and is no longer usable. In this case, the descriptor is put into a global recycle list (**DescriptorRetire()** in line 17) and the algorithm is repeated. Otherwise, all available blocks in the superblock are reserved, with a CAS that updates the anchor (line 23), and then added to the thread's cache (lines 24–29). No ABA-prevention tag is needed on this CAS, because the only change that can happen to the underlying superblock is that more blocks become available, thus updating **Anchor.avail**, which would fail the CAS. This is opposed to Michael's original algorithm, where blocks could concurrently become available and unavailable due to *active* superblocks.

If there are no available partial superblocks for the size class at hand, a new superblock must allocated and initialized. This is done in **CacheFillFromNewSB()**, which allocates a new superblock from the operating system and assigns a descriptor to it. The assigned descriptor is provided by **DescriptorAlloc()**, which accesses a global lock-free list of recycled descriptors. All blocks in the newly allocated superblock are then added to the thread's cache (lines 42–45

in Listing 1.4). Note that in low-concurrency scenarios, only a few CAS instructions are required to transfer a potentially large number of blocks to the cache. The exact number of CAS depends on which lock-free data structures are being used to track partial superblocks.

Listing 1.4. CacheFill routines

```
void CacheFill(size_t scIdx, Cache* cache) {
  // try to fill cache from a single partial superblock ...
  bool res = CacheFillFromPartialSB(scIdx, cache);
  // ... and if that fails, create a new superblock
  if (!res)
    CacheFillFromNewSB(scIdx, cache);
}

bool CacheFillFromPartialSB(size_t scIdx, Cache* cache) {
  Descriptor* desc = HeapGetPartialSB(scIdx);
  if (!desc)                    // no partial superblock available
    return false;
  Anchor newAnc, oldAnc;
  do {
    oldAnc = desc->anchor;
    if (oldAnc.state == EMPTY) {
      DescriptorRetire(desc);
      return CacheFillFromPartialSB(scIdx, cache);        // retry
    }
    newAnc.state = FULL;
    newAnc.avail = desc->maxcount;
    newAnc.count = 0;
  } while (!CAS(&desc->anchor, oldAnc, newAnc));
  char* block = desc->superblock + oldAnc.avail * desc->blockSize;
  size_t blockCount = oldAnc.count;
  while (blockCount-- > 0) {
    CachePushBlock(cache, block);
    block = *(char**)block;
  }
  return true;
}

void CacheFillFromNewSB(size_t scIdx, Cache* cache) {
  Descriptor* desc = DescriptorAlloc();
  DescriptorInit(desc, scIdx);  // initialize with size class info
  Anchor anc;
  anc.state = FULL;
  anc.avail = desc->maxcount;
  anc.count = 0;
  desc->anchor = anc;
  desc->superblock = mmap(...);
  for (size_t idx = 0; idx < desc->maxcount; ++idx) {
    char* block = desc->superblock + idx * desc->blockSize;
    CachePushBlock(cache, block);
  }
  PageMapRegisterDescriptor(desc);              // update pagemap
}
```

Flushing a cache is a less straightforward procedure. When a cache is full, several blocks must be returned to their respective superblocks, which requires updating the superblocks' associated descriptors. If each block was to be returned individually, a number of CAS instructions equal to the number of blocks to be removed would be required, which could be too inefficient and a source of contention. Instead, it is best to group up blocks according to descriptor as best as possible, in order to be able to return several blocks back to the same superblock in a single CAS. Listing 1.5 shows LRMalloc's cache flushing algorithm.

Listing 1.5. CacheFlush routine

```
1  void CacheFlush(size_t scIdx, Cache* cache) {
2    while (!CacheIsEmpty(cache)) {
3      // form a list of blocks to return to a common superblock
4      char* head, tail;
5      head = tail = CachePopBlock(cache);
6      Descriptor* desc = PageMapGetDescriptor(head);
7      size_t blockCount = 1;
8      while (!CacheIsEmpty(cache)) {
9        char* block = CachePeekBlock(cache);
10       if (PageMapGetDescriptor(block) != desc)
11         break;
12       CachePopBlock(cache);
13       ++blockCount;
14       *(char**)tail = block;
15       tail = block;
16     }
17
18     // add list to descriptor and update anchor
19     char* superblock = desc->superblock;
20     size_t idx = ComputeIdx(superblock, head);
21     Anchor oldAnc, newAnc;
22     do {
23       newAnc = oldAnc = desc->anchor;
24       *(char**)tail = superblock + oldAnc.avail * desc->blockSize;
25       newAnc.state = PARTIAL;
26       newAnc.avail = idx;
27       newAnc.count += blockCount;
28       if (newAnc.count == desc->maxcount)   // can free superblock
29         newAnc.state = EMPTY;
30     } while (!CAS(&desc, oldAnc, newAnc));
31
32     if (oldAnc.state == FULL)
33       HeapPutPartialSB(desc);
34     else if (newAnc.state == EMPTY) {
35       // unregister metadata from pagemap ...
36       PageMapUnregisterDescriptor(superblock, scIdx);
37       // ... and release superblock back to OS
38       munmap(superblock, ...);
39     }
40   }
41 }
```

CacheFlush() starts by popping a block from the cache and by forming an ad-hoc singly-linked list with all next blocks in cache that belong to the same superblock (lines 4 –16). Recall that blocks that belong to the same superblock share the same descriptor. We use **CachePeekBlock()** to inspect the first block in the cache without popping it. The ad-hoc list is then added to the corresponding superblock's available block list and the anchor updated accordingly (lines 19–30). At this stage, the CAS only fails if other blocks are being simultaneously flushed from other caches. At the end, if these are the first blocks to be freed, the superblock is added to the lock-free stack of partial superblocks (line 33). Otherwise, if all blocks are made free, the pagemap is updated to reflect the change and the superblock returned to the operating system (lines 35–38).

Note that a descriptor cannot be returned to the operating system since other threads can be potentially holding a reference to it. Moreover, it cannot be recycled also as there may still be a reference to it in the list of partial superblocks, and guaranteeing its correct removal is a non-trivial task. In our approach, a descriptor is only recycled when a thread removes it from the list of partial superblocks (line 17 in algorithm **CacheFillFromPartialSB()**).

3.3 Pagemap

The pagemap component stores *allocation metadata per page* based on the observation that most allocations are much smaller than a page, and that blocks in the same page share the same size class and descriptor. Storing metadata per page instead of per allocation has several advantages. First, it is more memory efficient. Second, it helps separating allocation metadata from user memory, which improves locality, as allocator and user memory are no longer interleaved.

The pagemap can be implemented in a number of different ways. With an operating system that allows memory overcommitting, it can be a simple array, where the size depends on the size of the valid address space and how much memory is needed for each page's metadata. For example, assuming that a single word of memory is enough for metadata, and that the address space can only have 2^{48} bytes (common for 64 bit architectures) then this array requires 2^{48-12} words, or about 512GB. Of course, the actual physical memory is bounded by the number of pages required for user applications. LRMalloc uses this type of implementation. A cheaper solution in terms of virtual memory would be to use a lock-free radix tree, at the cost of some performance and additional complexity.

4 Experimental Results

The environment for our experiments was a dedicated x86-64 multiprocessor system with four AMD SixCore Opteron TM 8425 HE @ 2.1 GHz (24 cores in total) and 128 GBytes of main memory, running Ubuntu 16.04 with kernel 4.4.0-104 64 bits. We used standard benchmarks commonly used in the literature [1, 11], namely the *Linux scalability* [10], *Threadtest* and *Larson* [7] benchmarks.

Linux scalability is a benchmark used to measure memory allocator latency and scalability. It launches a given number of independent threads, each of which runs a batch of 10 million **malloc()** requests allocating 16-byte blocks followed by a batch of identical **free()** requests.

Threadtest is similar to *Linux scalability* with a slightly different allocation profile. It also launches a given number of independent threads, each of which runs batchs of 100 thousand **malloc()** requests followed by 100 thousand **free()** requests. Each thread runs 100 batches in total.

Larson simulates the behavior of a long-running server process. It repeatedly creates threads that work on a slice of a shared array. Each thread runs a batch of 10 million **malloc()** requests between 16 and 128 bytes and the resulting allocations are stored in random slots in the corresponding slice of the shared array (each thread's slice includes 1000 slots). When a slot is occupied, a **free()** request is first done to release it. At any given time, there is a maximum limit of threads running simultaneously, i.e., only one thread has access to a given slice of the shared array, and slices are recycled as threads are destroyed and created.

To put our proposal in perspective, we compared LRMalloc against other memory allocators, such as, Michael's allocator [11], Hoard [1], Ptmalloc2 [5], Jemalloc-5.0 [3] and TCMalloc [4]. Figure 2 presents experimental results for the

benchmarks described above when using the different memory allocators with configurations from 1 to 32 threads. The results presented are the average of 5 runs.

Figure 2(a) shows the execution time, in seconds (log scale), for running the *Linux scalability* benchmark. In general, the results show that LRMalloc is very competitive, being only overtaken by Jemalloc as the number of threads increases. Jemalloc's behavior can be explained by the fact that it creates a number of arenas equal to four times the number of cores in the system. Multiple threads may be mapped to the same arena but, with so many arenas, collisions are unlikely and thus little contention happens on shared data structures. On the other hand, LRMalloc's heap has no arena-like multiplexing, and thus there is a greater potential for contention as the number of threads increases.

Figure 2(b) shows the execution time, in seconds (log scale), for running the *Threadtest* benchmark. Again, the results show that LRMalloc is very competitive and only surpassed by Jemalloc by a small margin. In particular, for a small number of threads, LRMalloc is slightly faster than Jemalloc and, as the number of threads increases, LRMalloc keeps an identical tendency as Jemalloc.

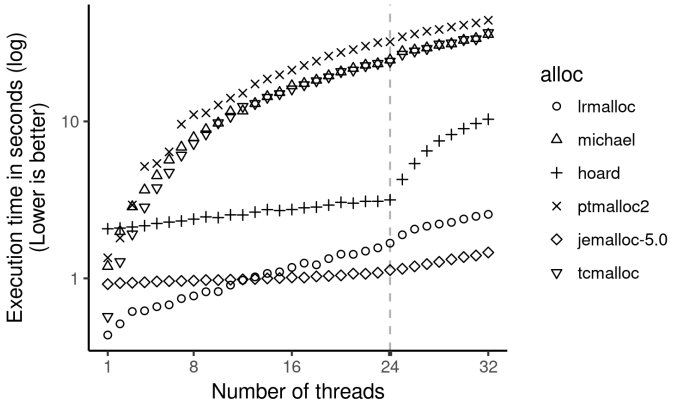
Figure 2(c) shows the number of operations per second (log scale), for running the *Larson* benchmark. In general, LRMalloc performs similarly to all the other allocators, with a small performance degradation as the number of threads increases. This happens due to the cache flushing mechanism triggered every-time a thread exits. This lowers memory fragmentation but, for the kind of programs which create a huge number of threads during its lifetime, it can incur in some extra overhead. This could be improved by recycling cache structures when a thread exits, in order to allow them to be reused by new spawning threads.

5 Conclusions and Further Work

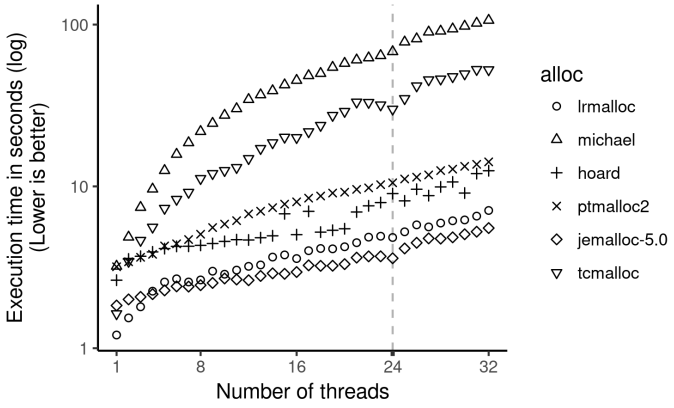
We have presented LRMalloc, a lock-free memory allocator designed to fulfill important and desirable properties in a memory allocator, such as, being immune to deadlocks and livelocks, and tolerant to arbitrary thread termination and priority inversion. It is our belief that future progress in memory allocators would involve the adoption of lock-free strategies as a way to provide these behavior properties to user applications.

Our experiments showed that LRMalloc's current implementation is already quite competitive and comparable to other modern state-of-the-art memory allocators. Nonetheless, there are a number of possible extensions that we plan to study which could further improve LRMalloc's performance. A first example is the support for multiple arenas as a way to reduce allocation contention as the number of threads increases, thus improving scalability. Another good example is the usage of improved cache management algorithms, as the ones implemented by TCMalloc [9], as a way to reduce the average memory allocator latency.

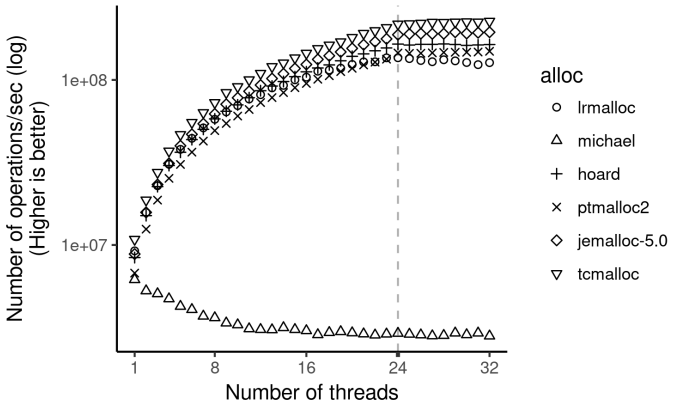
LRMalloc's current implementation uses a lock-free segregated storage scheme which relies on the operating system to provide continuous pages of memory, which we use to represent the superblocks. Another improvement we plan to



(a) Linux scalability



(b) Threadtest



(c) Larson

Fig. 2. Experimental results comparing Michael's allocator [11], Hoard [1], Ptmalloc2 [5], Jemalloc-5.0 [3] and TCMalloc [4] for the *Linux scalability*, *Threadtest* and *Larson* benchmarks with configurations from 1 to 32 threads

study is to have an additional lock-free component which handles these memory requests and is capable of general coalescing and splitting memory blocks representing the superblocks. To the best of our knowledge, no lock-free scheme exists that supports general coalescing and splitting. We believe that the results of such research could lead to relevant contributions able to be incorporated also into the operating system memory allocation system calls as a way to extend its lock-free properties.

Acknowledgements

We would like to thank the anonymous reviewers for their feedback and suggestions. Special thanks to Pedro Moreno for helpful technical discussion and ideas provided during the development of this research. This work is financed by the ERDF (European Regional Development Fund) through the Operational Programme for Competitiveness and Internationalisation - COMPETE 2020 Programme and by National Funds through the Portuguese funding agency, FCT (Portuguese Foundation for Science and Technology) within project POCI-01-0145-FEDER-016844.

References

1. Berger, E.D., McKinley, K.S., Blumofe, R.D., Wilson, P.R.: Hoard: A Scalable Memory Allocator for Multithreaded Applications. In: ACM SIGARCH Computer Architecture News. vol. 28, pp. 117–128. ACM (2000)
2. Dechev, D., Pirkelbauer, P., Stroustrup, B.: Understanding and Effectively Preventing the ABA Problem in Descriptor-based Lock-free Designs. In: 13th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing. pp. 185–192. IEEE (2010)
3. Evans, J.: A scalable concurrent malloc (3) implementation for FreeBSD. In: BSDCan Conference. Ottawa, Canada (2006)
4. Ghemawat, S., Menage, P.: Tcmalloc: Thread-caching malloc (2009), `http://goog-perftools.sourceforge.net/doc/tcmalloc.html` (read on June 14, 2018).
5. Gloger, W.: Ptmalloc (2006), `http://www.malloc.de/en` (read on June 14, 2018).
6. Hart, T.E., McKenney, P.E., Brown, A.D., Walpole, J.: Performance of memory reclamation for lockless synchronization. Journal of Parallel and Distributed Computing **67**(12), 1270–1285 (2007)
7. Larson, P.Å., Krishnan, M.: Memory Allocation for Long-Running Server Applications. ACM SIGPLAN Notices **34**(3), 176–185 (1998)
8. Lea, D.: A Memory Allocator Called Doug Lea's Malloc or dlmalloc for Short (1996), `http://g.oswego.edu/dl/html/malloc.html` (read on June 14, 2018).
9. Lee, S., Johnson, T., Raman, E.: Feedback Directed Optimization of TCMalloc. In: Workshop on Memory Systems Performance and Correctness. p. 3. ACM (2014)
10. Lever, C., Boreham, D.: malloc() Performance in a Multithreaded Linux Environment. In: USENIX Annual Technical Conference. pp. 301–311. USENIX (2000)
11. Michael, M.M.: Scalable Lock-Free Dynamic Memory Allocation. ACM Sigplan Notices **39**(6), 35–46 (2004)
12. Wilson, P.R., Johnstone, M.S., Neely, M., Boles, D.: Dynamic Storage Allocation: A Survey and Critical Review. In: Memory Management, pp. 1–116. Springer (1995)