A Lock-Free Coalescing-Capable Mechanism for Memory Management

Ricardo Leite
CRACS & INESC TEC and Faculty of Sciences
University of Porto
Portugal
rleite@dcc.fc.up.pt

Ricardo Rocha
CRACS & INESC TEC and Faculty of Sciences
University of Porto
Portugal
ricroc@dcc.fc.up.pt

Abstract

One common characteristic among current lock-free memory allocators is that they rely on the operating system to manage memory since they lack a lower-level mechanism capable of splitting and coalescing blocks of memory. In this paper, we discuss this problem and we propose a generic scheme for an efficient lock-free best-fit coalescing-capable mechanism that is able of satisfying memory allocation requests with desirable low fragmentation characteristics.

CCS Concepts • **Software and its engineering → Memory management**; *Allocation / deallocation strategies.*

Keywords Lock-Freedom, Memory Management, Allocation Mechanisms, Implementation, Evaluation.

ACM Reference Format:
Ricardo Leite and Ricardo Rocha. 2019. A Lock-Free Coalescing-Capable Mechanism for Memory Management. In *Proceedings of the 2019 ACM SIGPLAN International Symposium on Memory Management (ISMM '19), June 23, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3315573.3329982

1 Introduction

Memory allocation is a key component of most applications. Modern memory allocators are multi-layered pieces of software with demanding throughput and latency requirements. To meet these requirements, most of the layers in modern memory allocators are some form of intermediate cache services used to quickly respond to allocation and deallocation requests. Some examples are allocators based on thread-specific caches [5], arenas [4], slabs [2] or quick-lists [8]. However, at the lowest-level, all allocators require a component capable of managing memory – that is, capable of reusing memory that was once a block of some size to allocate another block of a different size. This task can be logically achieved with a component capable of *splitting* and *coalescing* blocks of memory (often blocks that are multiples of pages). Splitting is the act of dividing larger blocks of memory to fulfill smaller allocation requests. Coalescing is the act of joining adjacent free blocks into a single larger free block of memory. Splitting is done so that more memory remains available for future allocations. Coalescing is done so that future requests for larger blocks of memory can be fulfilled.

Several *coalescing-capable* mechanisms have been proposed in the past [23]. Among these are buddy systems, segregated fits and many forms of sequential fits (first-fit, next-fit, best-fit, etc). All of these mechanisms have been developed in the context of sequential execution, with a focus on reduced memory usage rather than memory request throughput or latency. As standalone memory allocators, these mechanisms have fallen out of favor and have been gradually replaced by concurrent memory allocators, with designs that increasingly use several layers. As a result, these mechanisms turned ineffective when used standalone, especially in a modern context where memory is cheap and plentiful and hardware has higher core counts. They are however still used as the lower-level mechanisms that provide memory for the higher-level cache services.

Modern memory allocators need to operate in concurrent environments. They thus need to employ synchronization primitives in order to handle concurrent memory allocation and deallocation requests. Nowadays, the most commonly used memory allocators are lock-based [4, 5, 7]. However, there have been some proposals for *lock-free* memory allocators [6, 14, 19, 22].

Lock-freedom is a desirable property for concurrent algorithms, as it 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 [10]. This progress guarantee is by far the most important advantage of lock-free synchronization and it protects the memory allocator (and therefore the running application) from reaching a complete halt even in the presence of a subpar operating system scheduler, or when a very high number of running

ISMM '19, June 23, 2019, Phoenix, AZ, USA

ACM ISBN 978-1-4503-6722-6/19/06…$15.00
https://doi.org/10.1145/3315573.3329982

threads exist when comparing to machine cores. By definition, lock-free algorithms use no locks and do not obtain mutual exclusive access to a resource at any point. They are 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. Instead of locks, lock-free algorithms use atomic instructions to guarantee consistency in concurrent environments. The most relevant atomic instruction is the CAS (Compare-and-Swap) instruction, which is widely supported in modern architectures.

One common characteristic among all proposed lock-free memory allocators is that they rely on the operating system to manage memory. An example of this is the allocation and deallocation of *superblocks*. Superblocks are continuous sets of pages that contains equal-sized blocks. Blocks of different sizes necessarily belong to different superblocks. In order for memory that belongs to a block B1 of size S1 to be used for a block B2 of size S2, the superblock that contains B1 must be freed back to the operating system, so that a superblock that contains blocks of size S2 can be allocated. The operating system is thus essential for the reuse of memory, and responsible of managing superblocks so that the rest of the allocator can function. Current proposals of lock-free memory allocators do not possess a lower-level coalescing-capable component able to take over this task, and are thus fundamentally unable of functioning in a environment where no operating system exists, or where the operating system only provides a single region of memory that has to be managed. On the other hand, modern lock-based memory allocators, such as ptmalloc2 [7], Jemalloc [4] and TCMalloc [5], all feature a coalescing-capable lower-level component. This has a number of advantages: (i) performance benefits, as less interaction with the operating system is required (usually through the use of the *mmap()* and *munmap()* system calls that are often prohibitively expensive); and (ii) practical implications, since this enables these memory allocators to be able to operate in the absence of an operating system, which can be important for their usage, for example, in embedded systems or as part of the operating system itself. In practice, a coalescing-capable lower-level component is used to manage blocks that are multiples of pages.

Remarkably, there is little research on lock-free versions of coalescing-capable mechanisms. The first design of a lock-free mechanism that jointly supports allocation, deallocation and coalescing operations is the recent work by Marotta et al. [16, 17], which proposes a lock-free binary buddy system named NBBS. However, the hierarchical structure that buddy systems impose is known to restrict the way in which coalescing can be performed and to cause high fragmentation in practice [11].

In this work, we propose a generic scheme for a lock-free best-fit coalescing-capable mechanism that supports lock-free splitting and coalescing of blocks with arbitrary sizes, and is equivalent to an address-ordered best-fit mechanism [23], which has desirable low fragmentation characteristics [24]. Our experiments, comparing the performance and scalability of our proposal against NBBS and the operating system, show that our proposal obtains the best results when using standard benchmarks commonly used in the literature, especially as the number of threads increases. We have modified these standard benchmarks to allocate/deallocate pages, rather than byte-sized blocks.

To the best of our knowledge, this is the first design of a generic lock-free coalescing-capable mechanism. Our mechanism does not put restrictions on how coalescing can be performed nor cause internal fragmentation due to the use of size classes. Ultimately, our aim is for our proposal to be used as a lower-level mechanism for lock-free memory allocators. In general, we argue that a lock-free coalescing-capable mechanism has value in the context of user-space lock-free memory allocators, allowing them to manage memory without the assistance of the operating system and thus improve performance by potentially reducing the number of system calls. A fully lock-free coalescing mechanism would furthermore allow practical usage of lock-free allocation in embedded systems and in environments where a single region of memory has to be managed. Moreover, we point out some drawbacks in our proposal, and leave it clear that there exists room for improvement in the subject of lock-free memory management. Among these drawbacks are the need for a secondary memory allocator in order to dynamically allocate nodes for internal data structures, and cases in which coalescing and allocation may fail due to temporary ownership of blocks. Due to these drawbacks, some assumptions are needed in order for our design to be lock-free.

The remainder of the paper is organized as follows. First, we discuss related work and present relevant background. Next, we describe the key concepts, support data structures, algorithms and open problems of our proposal. We then show an experimental analysis of our proposal. We end by outlining conclusions and discussing future work directions.

2 Related Work

We distinguish memory management from memory allocation, as memory allocation is often an efficient caching problem, especially as approached in the last decade, while memory management requires the capability to split and coalesce blocks of memory.

To be able to coalesce blocks, a coalescing-capable mechanism has to store information about what blocks are neighbors and whether those blocks are free and thus available for coalescing. Buddy systems [21], which impose a strict hierarchy on the address space, can quickly calculate the corresponding buddy block through a simple address computation, and thus only store a single bit of information per block to determine whether the buddy is free or is being used.

A more widely adopted mechanism is the usage of boundary tags [12] as originally proposed by Knuth. Boundary tags support splitting and coalescing of blocks with arbitrary sizes but are more memory-intensive, as they require having a header and a footer field per block containing information to track prior and next blocks. Since our proposal is also based on the concept of boundary tags, we discuss coalescing with boundary tags in more detail in the next section.

Regarding lock-free coalescing mechanisms, to the best of our knowledge, the only available design is a lock-free binary buddy system named NBBS [16, 17]. Classic binary buddy systems use a segregated fits mechanism containing one free list per each possible power of two size class of blocks. Blocks being coalesced are first removed from the corresponding free lists and the resulting coalesced block is then added. Instead of extending the segregated fits mechanism to support lock-freedom, NBBS uses a single statically allocated array containing a entry node for every *possible block* in the buddy system. In this array, nodes are organized by levels – starting with one node in level 1 for the single block of size 2^N, two nodes in level 2 for the 2 distinct blocks of size 2^{N-1}, and so forth – in the case of an address space of size 2^N and a minimum block size of 1, this equals to $2^{N+1}-1$ nodes in total.

In NBBS, each node corresponding to a non-leaf block (i.e., a block larger than the minimum size) has two child nodes that correspond to the two buddy blocks that each make up half of the block. Each node contains a number of flags with information regarding whether the block is free or occupied, and whether each child block is free, occupied or undergoing coalescing. An allocation is performed by linearly traversing the nodes fitting the level corresponding to the requested size. If a free node is found, it is marked as occupied, and then all ancestor nodes are also marked as having an occupied child. In case an occupied ancestor is found, all the markings are undone and the searching for a valid node continues. In the deallocation case, ancestor nodes are marked as having a child that is undergoing coalescing. After all ancestors are marked, the node is marked as free, and coalescing and occupied child flags are removed from ancestors.

In Sec. 5, we show experimental results comparing NBBS against our proposal.

3 Coalescing with Boundary Tags

Consider that we have a block that has been deallocated and that we want to coalesce it with its neighbors. To perform coalescing, we need block information allowing to locate its neighbors and their state. This section discusses how generic coalescing can be performed with boundary tags [12].

Basics

With boundary tags, all blocks have an associated *header* and *footer*. The header contains information about the block, namely its size and its state (whether it is currently allocated or free). The footer contains a back pointer to the header of the same block that it belongs to.[1] Figure 1 illustrates this block information.

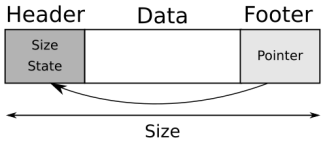
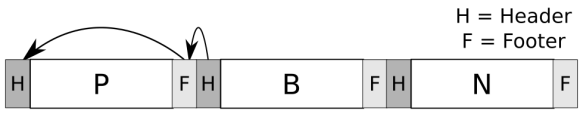
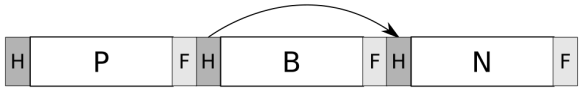


Figure 1. Block information needed for coalescing

Every block B has two neighbors that it might be able to coalesce with, the one that precedes B in memory and the one that follows B in memory. As Fig. 2 shows, given a block B, we can locate the previous block P by reading the memory that immediately precedes B's header, which corresponds to P's footer, and then follow the footer's back pointer to locate P's header, where state information is stored. Similarly, we can locate the next block N by using B's size to locate N's header. We name the process of block coalescing with the previous and the next block, respectively, as *backward coalescing* and *forward coalescing*.



(a) Locating the header of the previous block



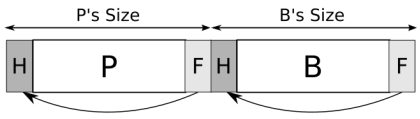
(b) Locating the header of the next block

Figure 2. Locating the previous and next blocks

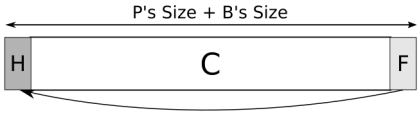
Once we have located a block, we can begin to attempt to coalesce. With boundary tags, that corresponds to building a new block with the respective header and footer. Figure 3 shows an example of backward coalescing where a block B is being coalesced with a previous block P to form a new coalesced block C. In such case, P's header becomes C's header, B's footer becomes C's footer, and P's footer and B's header become irrelevant.

Analogously, in forward coalescing, where a block B coalesces with a next block N to form a new coalesced block C, B's header becomes C's header, N's footer becomes C's footer, and B's footer and N's header become irrelevant. In both backward and forward coalescing, C's header and footer have to be updated and adjusted to C's size, thus storing different information than the original blocks.

[1] Note that this back pointer can be also implemented using the block size, as in the case of the header.



(a) Before coalescing



(b) After coalescing

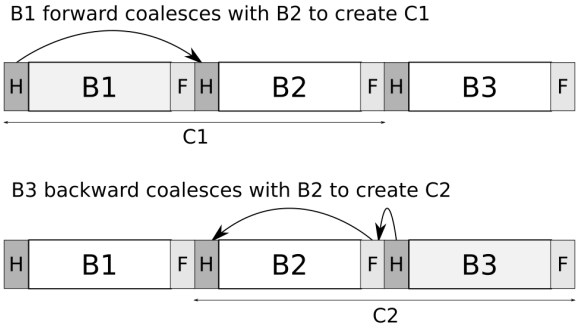
Figure 3. Backward coalescing

Lock-Freedom

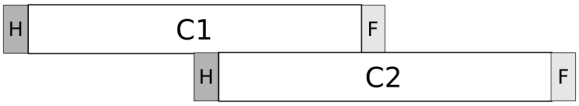
Performing coalescing with boundary tags in a sequential environment is trivial. In a multithreaded environment, it is similarly trivial if a synchronization lock is used to serialize concurrent coalescing operations. Current state-of-the-art lock-based memory allocators perform concurrent coalescing by defining multiple regions, each with a single lock.

Compared to lock-based approaches, lock-free coalescing presents a number of challenges. First, with boundary tags, we have two distinct pieces of information – the header and the footer of each block – that ideally would have to be kept coherent, but which cannot be ensured if using single CAS instructions. Second, we cannot guarantee that, at some time interval, no other thread changes the state of a neighbor block that we are trying to coalesce with. Third, the region of memory being managed is finite, thus not all blocks have neighbors in both directions. Furthermore, since such region might increase and decrease over time, when reading the memory that precedes/follows a given block, we might end reading invalid memory that we are not managing. In a sequential or serialized multithreaded execution, it is trivial to check the limits of the region of memory being managed. However, in a lock-free coalescing mechanism, nothing ensures that such limits do not change between the moment at which we read them and the moment at which we read the memory that precedes/follows a block, thus potentially reading invalid memory.

To better understand the difficulty of lock-free coalescing, consider the example in Fig. 4 where two blocks $B1$ and $B3$ are deallocated simultaneously and attempt to coalesce with a free block $B2$, which sits between $B1$ and $B3$. With boundary tags, coalescing involves locating the header of the neighbor block, checking whether the neighbor block is free for coalescing, and atomically updating the header, then the footer, for the new coalesced block. As seen in Fig. 4a, both $B1$ and $B3$ can reach $B2$'s header to attempt coalescing. However, because building a new coalesced block is a two-step process, $B1$ can locate $B2$, begin coalescing, update the header for the new coalesced block $C1$ (which is $B1$'s header), and then $B3$ can locate $B2$ (through $B2$'s footer) and successfully coalesce and create block $C2$ before $B1$ has a chance to update $B2$'s footer. The result of this simultaneous coalescing are two incorrect overlapping blocks $C1$ and $C2$, as shown in Fig. 4b.



(a) Blocks $B1$ and $B3$ simultaneously locate block $B2$ to attempt coalescing



(b) Blocks $B1$ and $B3$ can both succeed, with a specific order of operations, creating blocks $C1$ and $C2$, which overlap

Figure 4. Simultaneous coalescing with a shared neighbor

Note that, so far, we have only discussed how coalescing is done once we have a block that has been deallocated and that we wish to coalesce with its neighbors. We have not discussed how free blocks are tracked nor how blocks are allocated in the first place. The use of additional data structures, needed to find and allocate blocks, naturally complicates the goal of designing a lock-free coalescing mechanism.

4 Our Proposal

This section presents and discusses the design and implementation issues of our lock-free coalescing proposal.

4.1 Support Data Structures

Our proposal for a lock-free best-fit mechanism consists of two logical data structures: (i) the *mapper*, which represents the physical layout in memory; and (ii) the *controller*, an ordered set-like data structure used to store free blocks in a best-fit fashion. All blocks include a header and a footer, to be managed by the mapper. Both the header and the footer simply store the corresponding block's size. Neither the header nor the footer contain information on whether the block is allocated or free. Instead, the controller is the authority used to assert whether a block is allocated or free. The controller is simultaneously used to satisfy allocations, but also to determine whether coalescing can be done. Before using a block for coalescing or to fulfill an allocation, the controller is used to ensure that a single thread obtains the block's ownership. Indeed, the main insight of our proposal is that

the data structure used to obtain free blocks can also be the one used to coordinate coalescing. Otherwise, if state is kept elsewhere (i.e., in the header, in the footer or in other auxiliary data structure), the mechanism to determine whether a block is free for coalescing inevitably becomes unable to keep both the data structure and the referred state coherent with each other when coalescing occurs.

Controller

The controller is an indexing data structure that manages free blocks. In order to implement best-fit, it sorts blocks by size, and blocks with the same size by address. In our current implementation, the controller is implemented using a lock-free binary tree proposed by Natarajan and Mittal [20]. A block is represented by its starting address and by its size. Both the starting address and size are multiples of the *block unit* being managed, e.g., to handle page block allocations, a 1-page block unit is used. The operations supported by the controller are shown next in Listing 1.

Listing 1. Controller operations

```
1  struct block {
2   void* addr;  // block's starting address
3   size_t size; // block's size
4  };
5
6  // add block to controller
7  void ControllerAddBlock(block b);
8
9  // try to remove block from controller
10 bool ControllerRemoveBlock(block b);
11
12 // get block from controller given a size
13 block ControllerGetBlock(size_t size);
```

The controller is implemented as a lock-free indexed data structure. Because it relies on the lock-freedom of the used data structure, we next describe in more detail the expected behavior and requirements of each operation, so that our coalescing scheme works as intended. The **ControllerAddBlock()** routine adds a block to the controller that must be available for removal when the routine returns. It can be assumed that the given block does not yet exist in the controller. The **ControllerRemoveBlock()** routine removes a given block from the controller, if it exists, and returns true. Otherwise, it returns false. The removal must be complete when the routine returns, such that the same block can be immediately added back to the controller. The **ControllerGetBlock()** routine removes and returns the least-ordered block from the controller that satisfies the requested size. If no block is found, it returns an invalid block starting at address 0 and with size 0. This routine has the same removal requirements as **ControllerRemoveBlock()**.

The routines **ControllerAddBlock()** and **ControllerRemoveBlock()** map directly into the add and remove operations of any lock-free data structure. The **ControllerGetBlock()** routine is specific to our application and does not map directly into the usual operations available in lock-free data structures. We have extended the lock-free binary tree proposed by Natarajan and Mittal [20] to support it. In our implementation, the **ControllerGetBlock()** routine traverses the tree until it finds a block satisfying the requested size. If a block is not found in a traversal, the requested size is increased to a size found in one of the internal routing nodes of the tree. The traversal is then repeated from the beginning until the size exceeds the size of the largest available block or a suitable block is found.

Mapper

The mapper represents the physical layout in memory, and allows a block to find its neighbors in order to perform coalescing. Each block logically has a header and a footer that can be accessed by neighboring blocks. However, to allow the mechanism to manage varying amounts of memory and to avoid invalid accesses to memory, as discussed earlier, a block does not physically contain its header and its footer. In essence, the header and the footer are persistent objects belonging to the mapper, which are reused as blocks are being split and coalesced. The operations that the mapper needs to support are shown next in Listing 2.

Listing 2. Mapper operations

```
1  struct header { size_t size; };
2  struct footer { size_t size; };
3
4  // get block from mapper given an address
5  block MapperGetBlock(void* addr) {
6   header h = GetHeader(addr);
7   block b = {addr, h.size};
8   return b;
9  }
10
11 // get block before a given block
12 block MapperGetPreviousBlock(block b) {
13  footer f = GetFooter(b.addr - 1);
14  block p = {b.addr - f.size, f.size};
15  return p;
16 }
17
18 // get block after a given block
19 block MapperGetNextBlock(block b) {
20  header h = GetHeader(b.addr + b.size);
21  block n = {b.addr + b.size, h.size};
22  return n;
23 }
24
25 // update mapper with a given block
26 void MapperUpdate(block b) {
27  UpdateHeader(b.addr, b.size);
28  UpdateFooter(b.addr + b.size - 1, b.size);
29 }
```

For all the addresses within the same block unit, the **GetHeader()** and **GetFooter()** routines return the same corresponding header and footer, respectively. The same idea applies to the first argument of the **UpdateHeader()** and **UpdateFooter()** routines. Both the header and the footer are small enough to fit inside a single processor word, and thus they can be atomically updated through CAS atomic

operations. Therefore, both **UpdateHeader()** and **UpdateFooter()** routines are equivalent to an atomic write operation, while **GetHeader()** and **GetFooter()** are equivalent to atomic reads.

A possible implementation for the mapper is to use an array of uncommitted memory with two processor words (header and footer) per block unit (i.e., per page if assuming a 1-page block unit). In fact, in our implementation this is further optimized to one processor word per page. An alternative implementation for the mapper would be to use a lock-free radix-tree.

4.2 High-level Allocation and Deallocation

Next, we show how the high-level allocation and deallocation routines interact with the mapper and the controller.

Listing 3 shows the high-level allocation routine. It starts by trying to get a block with sufficient size from the controller (line 2) and, if no adequate block is found, more memory has to be obtained from the operating system and a block constructed with it.[2] Otherwise, if a block is found, it can be used to fulfill the memory request. To eliminate internal fragmentation, the block is split if it is larger than what has been requested (lines 6–17). Splitting consists of updating the header and the footer for the original and the remaining free block, and then adding the remaining free block to the controller, so that it may be used to fulfill further memory allocation requests or for coalescing.

Listing 3. High-level allocation

```
1  void* Alloc(size_t size) {
2    block b = ControllerGetBlock(size);
3    if (b.addr == NULL) { // no block was found
4      ... // obtain more memory or fail
5    }
6    if (b.size > size) { // too large, split
7     // at this point the block is owned by us,
8     // so it can be manipulated safely
9     block s = {b.addr, size};
10    MapperUpdate(s);
11    // remaining free block
12    block r = {b.addr + size, b.size - size};
13    MapperUpdate(r);
14    // add remaining free block to controller,
15    // so that it becomes available
16    ControllerAddBlock(r);
17   }
18   return b.addr;
19 }
```

Listing 4 shows the high-level deallocation routine. Deallocation starts by finding the block corresponding to the given address (line 2). Then, since the block is owned by us, we can attempt to coalesce this block with both neighbors (lines 4 and 6). Coalescing is further detailed next. After we have attempted to coalesce, we can add the resulting block to the controller (line 8).

[2] In environments where a single region of memory has to be managed and no more memory exists, the execution simply fails accordingly to some pre-defined errors.

Listing 4. High-level deallocation

```
1  void Dealloc(void* addr) {
2    block b = MapperGetBlock(addr);
3    // attempt coalescing with previous block
4    b = CoalesceBackward(b);
5    // attempt coalescing with next block
6    b = CoalesceForward(b);
7    // add (coalesced) block to controller
8    ControllerAddBlock(b);
9  }
```

4.3 Coalescing

Coalescing only occurs when a block is in the process of being deallocated, which means that the block is not yet stored in the controller. To coalesce a deallocated block, we have to locate its neighbors with the mapper and then attempt to remove each of those neighbors from the controller. Remember that such removal from the controller is only possible if the neighbors blocks are free blocks. If we can remove a neighbor block, we have acquired ownership of it, and thus we can use it for coalescing. Both backward and forward coalescing operate in the same manner, by attempting to remove the previous/next block from the controller, and then performing coalescing.

Listing 5 shows the backward coalescing routine. Backward coalescing begins by locating the previous block in the mapper (line 3) and then by attempting to remove it from the controller (line 4). Note that the controller stores not only block sizes or addresses, but both. This is important, as we could otherwise be removing a block that starts at the previous block's address, but has a different size (thus not sharing a boundary with our block). If the removal of the previous block is successful, we can perform coalescing and update the corresponding header and footer in the mapper (line 8).

Listing 5. Backward coalescing

```
1  block CoalesceBackward(block b) {
2   // get previous block
3   block p = MapperGetPreviousBlock(b);
4   if (ControllerRemoveBlock(p)) {
5    // previous block is free, can coalesce
6    b = {p.addr, p.size + b.size};
7    // update mapper to reflect change
8    MapperUpdate(b);
9   }
10  return b;
11 }
```

Listing 6 shows the forward coalescing routine. Forward coalescing follows the same pattern as backward coalescing. The key difference is that, instead of locating the previous block, it begins by locating the next block in the mapper (line 3). Then, it also attempts to remove the next block from the controller (line 4) and, if successful, it performs coalescing and updates the corresponding header and footer in the mapper (line 8).

Listing 6. Forward coalescing

```
1  block CoalesceForward(block b) {
2   // get next block
3   block n = MapperGetNextBlock(b);
4   if (ControllerRemoveBlock(n)) {
5    // next block is free, can coalesce
6    b = {b.addr, b.size + n.size};
7    // update mapper to reflect change
8    MapperUpdate(b);
9   }
10  return b;
11 }
```

Note that the high-level allocation and deallocation routines, and therefore the backward and forward coalescing routines, are trivially lock-free if the controller is implemented with a lock-free data structure.

4.4 Open Problems

There are some caveats in our current proposal and implementation that we would like to point out.

A first problem is that our design cannot guarantee that coalescing always occur. Consider an example with two neighboring blocks, B1 and B2, owned by threads T1 and T2, respectively. If T1 and T2 deallocate B1 and B2 simultaneously, T1 will try to remove B2 from the controller, and T2 will try to remove T1, but both will fail. Then T1 will add B1 to the controller and T2 will add B2. We thus end up with two adjacent blocks, B1 and B2, which are free but not coalesced. This can be mitigated by performing *recursive coalescing*, e.g., continue doing backward and forward coalescing while there are adjacent free blocks. In the context of the example, that ensures that both B1 and B2 will be coalesced when another neighbor block is deallocated and attempts to coalesce. The failure of coalescing reveals an interesting property – as contention increases and more threads operate on the same space, the more likely it is that coalescing fails. In Sec. 5, we experimentally measure how often coalescing fails.

Another caveat is that the controller is implemented by a lock-free data structure that likely uses nodes that must be dynamically allocated. In particular, the data structure we have chosen for our implementation [20] needs dynamic allocation of nodes, and furthermore, is vulnerable to the ABA problem [3] without the use of a memory reclamation method such as epochs or hazard pointers [9, 18]. Dynamically allocating memory for a data structure inside a memory allocator is naturally tricky. It requires our design to assume the existence of a secondary memory allocator capable of allocating equal-sized nodes that must also be lock-free. While the allocation of equal-sized nodes is an easier problem than general memory allocation, it complicates the implementation and practicality of our proposal. For our design to be used as a lower-level allocator in a production memory allocator, such as TCMalloc [5] or Jemalloc [4], the controller component would ideally not need to dynamically allocate memory, and thus use a completely static data structure. We leave it as further work whether there is a lock-free data structure that is better suited to this context.

Finally, in order to perform splitting and coalescing, our proposal requires that threads acquire temporarily ownership of the blocks involved in such operations. While this is what allows splitting and coalescing to be performed in the first place, it can also cause allocation requests from other threads to fail, if they could not use the free space in the blocks that have been temporarily acquired. Consider a case in which the controller contains a single block B1 that has 10 pages, and threads T1 and T2 that are simultaneously attempting to allocate 1-page blocks. In order for T1 to allocate a 1-page block, it must remove B1 from the controller (thus acquiring temporary ownership), and then split B1 and add the remainder block B2 back to the controller (thus releasing temporary ownership). While T1 has temporary ownership, T2 will be unable remove any block from the controller and thus unable to allocate another 1-page block. Indeed, an unbounded number of allocations may fail until B2 is added back to the controller, which compromises our proposal's ability to allocate in a lock-free manner, especially without assuming another underlying allocator that can satisfy failed allocation requests (i.e., the OS).

A possible workaround to attenuate this problem is to set a maximum block size, such that, coalescing is simply not performed after a block grows sufficiently large. The controller will therefore have a larger number of blocks, of which, each thread may temporarily hold at most one. Thus, the use of our mechanism is unsuitable in environments without an OS, but otherwise viable in the context of a lock-free user-space memory allocator.

5 Experimental Analysis

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 4.4.0-141 64 bits. To measure the performance of our mechanism, we used standard benchmarks commonly used in the literature [1, 13, 15, 19], namely the *Linux scalability*, *Threadtest* and *Larson* benchmarks. Although these benchmarks are used to evaluate memory allocators, we believe that they are also a good fit to be used to evaluate a coalescing-capable mechanism. We have adapted these benchmarks to request pages rather than smaller byte-sized blocks.

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 thousand allocation requests allocating 1-page blocks followed by a batch of identical deallocation 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 batches of 100 allocation requests allocating 1-page blocks followed by 100 deallocation 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 100 allocation requests between 1 and 128 pages and the resulting allocations are stored in random slots in the corresponding slice of the shared array (each thread's slice includes 100 slots). When a slot is occupied, a deallocation 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. This benchmark runs for a total of 5 seconds, and measures number of allocations performed per second rather than execution time.

We have changed these benchmarks such that they *do not touch* the pages being provided in the allocation requests. This eliminates any overhead due to page faults and physical memory allocation, which would be a constant factor regardless of the mechanism being used.

Performance Evaluation

To put our proposal in perspective, we compared our mechanism against the NBBS[3] lock-free buddy system and against the operating system. We used our mechanism with and without recursive coalescing. Unlike NBBS, our implementation needs dynamic allocation of nodes for the lock-free data structure used in the controller. To evaluate the operating system, we implemented a lower-level allocator that uses the *mmap()* and *munmap()* system calls to allocate and deallocate pages.

Figures 5 to 7 present experimental results for the benchmarks described above when using the different mechanisms with configurations from 1 to 32 threads (note that the hardware used only supports 24 native threads). The results presented are an average of 5 runs.

Figure 5 shows the execution time, in seconds (log scale), for running the *Linux scalability* benchmark. The results show that our mechanism obtains the best results, with a performance tendency comparable to the operating system, as the number of threads increases. We note that NBBS performs quite badly, even with a single thread, and suffers from massive performance degradation, as the number of threads increases. Upon inspection of NBBS's source code, we observed that the core allocation algorithm requires traversing all allocated blocks in cases where blocks are allocated sequentially without interleaving deallocations. In essence, this case leads to a $O(n^2)$ expected runtime, where n is the number of allocated blocks. We highlight that these results differ from the original paper in [16]. We have found that the *Linux-scalability* benchmark [15] was not correctly implemented by the authors, and the results shown are of a different allocation pattern – n allocation/deallocation pairs, rather than a batch of n allocations followed by a batch of n deallocations, as in the original *Linux-scalability* benchmark.

[3]Downloaded from https://github.com/HPDCS/NBBS in January 4, 2019.

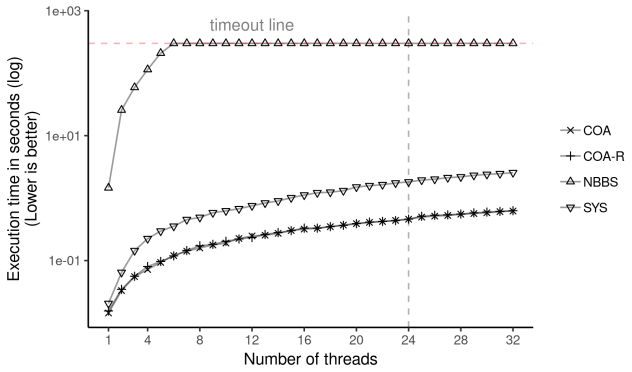


Figure 5. Execution time results, in seconds (log scale), comparing our mechanism, with (COA-R) and without (COA) recursive coalescing, the lock-free buddy system (NBBS) and the operating system (SYS) for the *Linux scalability* benchmark with configurations from 1 to 32 threads

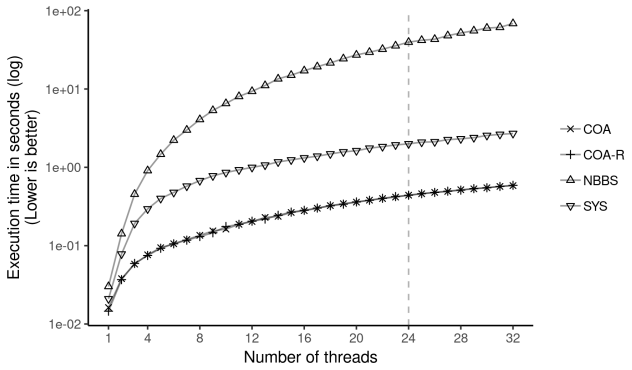


Figure 6. Execution time results, in seconds (log scale), comparing our mechanism, with (COA-R) and without (COA) recursive coalescing, the lock-free buddy system (NBBS) and the operating system (SYS) for the *Threadtest* benchmark with configurations from 1 to 32 threads

Figure 6 shows the execution time, in seconds (log scale), for running the *Threadtest* benchmark. Similarly to the previous benchmark, our mechanism performs quite nicely, with execution times one order of magnitude better than the operating system and two orders of magnitude better than NBBS. Similarly to the previous benchmark, all mechanisms show a performance degradation as the number of threads increases.

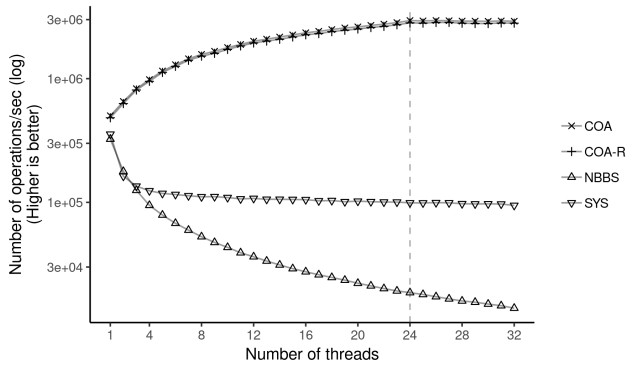


Figure 7. Throughput results, in seconds (log scale), comparing our mechanism, with (COA-R) and without (COA) recursive coalescing, the lock-free buddy system (NBBS) and the operating system (SYS) for the *Larson* benchmark with configurations from 1 to 32 threads

Figure 7 shows the number of operations per second (log scale) for running the *Larson* benchmark. Note that, unlike previous figures, this benchmark measures allocation throughput, and thus higher Y-axis values are better. Results show that our mechanism is capable of consistently handling higher allocation throughput as the number of threads increases, which is not the case with NBBS and the operating system. This is a highly desirable behavior in a coalescing-capable mechanism. This can be explained by the allocation pattern in the *Larson* benchmark, where allocations of different sizes are made, which leads to searches and modifications in different parts of the lock-free tree (controller) used in our implementation, thus leading to decreased contention points.

In all experiments, the results obtained for our mechanism with and without recursive coalescing are very similar, which shows that the cost of recursive coalescing is negligible for these benchmarks.

Coalescing Failures

Section 4.4 describes the problem of coalescing failure in cases where two threads deallocating neighboring blocks attempt to perform coalescing simultaneously. To measure the effect of this problem, we have designed a benchmark to test how frequently coalescing failures occur in an extreme high-throughput scenario where all threads are simultaneously deallocating blocks. Prior to launching any threads, the benchmark configures our coalescing mechanism to manage a single region of memory, corresponding to a single block in the controller. This region contains enough memory to include a fixed number B of allocated blocks to be assigned to each of the T threads being considered (for a total of $T * B$ blocks). These blocks are randomly distributed to the threads. Then, T threads are launched simultaneously, and each thread deallocates the B blocks assigned to it. At the end, the number of remaining blocks in the controller dictates the number of coalescing operations that have failed due to simultaneous coalescing. If a single block remains in the controller, that means that no coalescing failure occurred.

Figure 8 shows the frequency of coalescing failures using our mechanism with recursive coalescing turned off for a varying number of threads and a varying number of blocks assigned to each thread. The results presented are an average of 1000 runs.

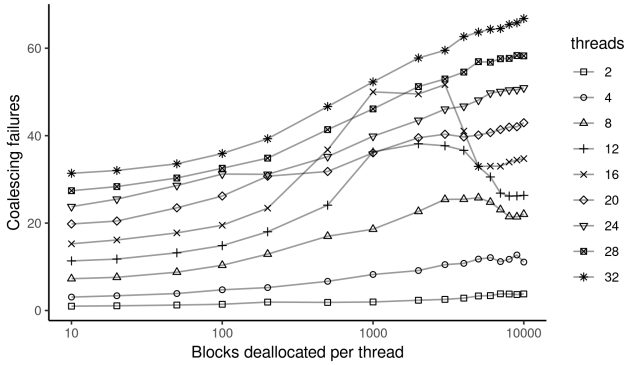


Figure 8. Failed coalescing attempts when using our mechanism without recursive coalescing in a high-throughput scenario with configurations from 2 to 32 threads

In general, we can observe that the number of coalescing failures increases as the number of blocks to be deallocated increases. However, the coalescing failures only occur in a very small percentage of all deallocation operations, such that, the total number of coalescing failures is closer to the number of threads itself. For the experiments with configurations between 500 and 4000 blocks allocated and with 8, 12 and 16 threads, there is an higher amount of coalescing failures, which can be explained by the fact that, for some reason, there is higher contention in the deallocation of the last blocks. Anyway, this benchmark shows that coalescing failure is relatively infrequent, even in an extremely high-throughput scenario, which is unrealistic to occur in practice. We can then conclude that, in real-world scenarios, this type of coalescing failure will rarely occur.

We also tested this benchmark using our mechanism with recursive coalescing turned on and we observed that, on average, the number of remaining blocks in the controller is close to 1. Only in a very small fraction of the experiments, the controller ends with 2 or 3 blocks maximum.

6 Conclusion

We have presented the first known design of a generic lock-free coalescing-capable mechanism. Our mechanism is capable of splitting and coalescing blocks with arbitrary sizes, i.e., without inducing internal fragmentation, and is equivalent to an address-ordered best-fit mechanism, which has desirable

low fragmentation characteristics. We have described the key concepts, design decisions, implementation difficulties and challenges of our proposal.

Our experiments, comparing the performance and scalability of our proposal against NBBS and the operating system, show that our proposal obtains the best results when using standard benchmarks commonly used in the literature, especially as the number of threads increases. We also observed that coalescing failure is relatively infrequent, even in extremely high-throughput scenarios.

Further work includes study whether there is a lock-free data structure that is better suited for usage in our proposal and the design of new schemes capable of attenuating or solving the open problems described previously.

Acknowledgments

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 projects UID/EEA/50014/2019 and POCI-01-0145-FEDER-016844.

References

[1] Emery D Berger, Kathryn S McKinley, Robert D Blumofe, and Paul R Wilson. 2000. Hoard: A Scalable Memory Allocator for Multithreaded Applications. In *ACM SIGARCH Computer Architecture News*, Vol. 28. ACM, 117–128.

[2] Jeff Bonwick. 1994. The Slab Allocator: An Object-Caching Kernel Memory Allocator. In *USENIX summer*, Vol. 16. Boston, MA, USA.

[3] Damian Dechev, Peter Pirkelbauer, and Bjarne Stroustrup. 2010. 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*. IEEE, 185–192.

[4] Jason Evans. 2006. A scalable concurrent malloc (3) implementation for FreeBSD. In *BSDCan Conference*.

[5] Sanjay Ghemawat and Paul Menage. 2009. TCMalloc: Thread-caching malloc. http://goog-perftools.sourceforge.net/doc/tcmalloc.html (read on June 14, 2018).

[6] Anders Gidenstam, Marina Papatriantafilou, and Philippas Tsigas. 2010. NBmalloc: Allocating Memory in a Lock-Free Manner. *Algorithmica* 58, 2 (2010), 304–338.

[7] Wolfram Gloger. 2006. Ptmalloc. http://www.malloc.de/en (read on June 14, 2018).

[8] Mel Gorman. 2004. *Understanding the Linux Virtual Memory Manager*. Prentice Hall Upper Saddle River.

[9] Thomas E Hart, Paul E McKenney, Angela Demke Brown, and Jonathan Walpole. 2007. Performance of memory reclamation for lockless synchronization. *J. Parallel and Distrib. Comput.* 67, 12 (2007), 1270–1285.

[10] Maurice Herlihy and Nir Shavit. 2011. On the Nature of Progress. In *International Conference On Principles Of Distributed Systems*. Springer, 313–328.

[11] Mark S Johnstone and Paul R Wilson. 1998. The Memory Fragmentation Problem: Solved?. In *ACM SIGPLAN Notices*, Vol. 34. ACM, 26–36.

[12] Donald E Knuth. 1997. *The Art of Computer Programming*. Vol. 3. Pearson Education.

[13] Per-Åke Larson and Murali Krishnan. 1998. Memory Allocation for Long-Running Server Applications. *ACM SIGPLAN Notices* 34, 3 (1998), 176–185.

[14] Ricardo Leite and Ricardo Rocha. 2018. LRMalloc: a Modern and Competitive Lock-Free Dynamic Memory Allocator. In *13th International Meeting on High Performance Computing for Computational Science (VECPAR 2018)*.

[15] Chuck Lever and David Boreham. 2000. malloc() Performance in a Multithreaded Linux Environment. In *USENIX Annual Technical Conference*. USENIX, 301–311.

[16] Romolo Marotta, Mauro Ianni, Andrea Scarselli, Alessandro Pellegrini, and Francesco Quaglia. 2018. A Non-blocking Buddy System for Scalable Memory Allocation on Multi-core Machines. *CoRR* abs/1804.03436 (2018).

[17] Romolo Marotta, Mauro Ianni, Andrea Scarselli, Alessandro Pellegrini, and Francesco Quaglia. 2018. A Non-blocking Buddy System for Scalable Memory Allocation on Multi-core Machines. In *2018 IEEE International Conference on Cluster Computing*. IEEE, 164–165.

[18] Maged M Michael. 2004. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Transactions on Parallel & Distributed Systems* 15, 6 (2004), 491–504.

[19] Maged M Michael. 2004. Scalable Lock-Free Dynamic Memory Allocation. *ACM Sigplan Notices* 39, 6 (2004), 35–46.

[20] Aravind Natarajan and Neeraj Mittal. 2014. Fast Concurrent Lock-free Binary Search Trees. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Vol. 49. ACM, 317–328.

[21] James L. Peterson and Theodore A. Norman. 1977. Buddy Systems. *Commun. ACM* 20, 6 (1977), 421–431.

[22] Sangmin Seo, Junghyun Kim, and Jaejin Lee. 2011. SFMalloc: A Lock-Free and Mostly Synchronization-Free Dynamic Memory Allocator for Manycores. In *2011 International Conference on Parallel Architectures and Compilation Techniques*. IEEE, 253–263.

[23] Paul R Wilson, Mark S Johnstone, Michael Neely, and David Boles. 1995. Dynamic Storage Allocation: A Survey and Critical Review. In *Memory Management*. Springer, 1–116.

[24] Paul R Wilson, Mark S Johnstone, Michael Neely, and David Boles. 1995. *Memory Allocation Policies Reconsidered*. Technical Report. Technical report, University of Texas at Austin Department of Computer Sciences.