NBBS: A Non-Blocking Buddy System for Multi-core Machines

Romolo Marotta, Mauro Ianni, Alessandro Pellegrini and Francesco Quaglia

Abstract—Common implementations of core memory allocation components handle concurrent allocation/release requests by synchronizing threads via spin-locks. This approach is not prone to scale, a problem that has been addressed in the literature by introducing layered allocation services or replicating the core allocators—the bottom-most ones within the layered architecture. Both these solutions tend to reduce the pressure of actual concurrent accesses to each individual core allocator. In this article, we explore an alternative approach to scalability of memory allocation/release, which can be still combined with those literature proposals. We present a fully non-blocking buddy system, where threads performing concurrent allocations/releases do not undergo any spin-lock based synchronization. Our solution allows threads to proceed in parallel, and commit their allocations/releases unless a conflict is materialized while handling the allocator metadata—memory fragmentation and coalescing are also carried out in a fully non-blocking manner. Conflict detection relies in our solution on atomic Read-Modify-Write (RMW) machine instructions, guaranteed to execute atomically by the processor firmware. We also provide a proof of the correctness of our non-blocking buddy system and show the results of an experimental study that outlines the effectiveness of our solution.

Index Terms—Non-blocking algorithms, Shared-memory algorithms, Back-end memory allocation, Buddy system

1 Introduction

In standard libraries or Operating Systems (OSs), memory allocation is de-facto a *shared-data management* problem. In fact, concurrent threads can issue allocation/release requests to a same memory allocator. This requires thread-coordination mechanisms in order to guarantee that the state of the memory allocator is always maintained coherent.

Commonly, thread coordination has been implemented as a spin-lock protected critical section, which can hamper scalability especially at larger thread counts. This is a relevant issue, since the level of thread-concurrency is increasingly exacerbated because of the modern-hardware trend towards multi/many-core technologies.

Literature studies have tackled the reduction of the impact of thread coordination (and the associated costs) on performance via: (a) pre-reserving *arenas*, namely memory segments, for each individual thread—this is what typically happens in user-space allocators [1]—or (b) the usage of intermediate allocation services, called *cached allocators*—as for the case of OS-kernel allocation services based on quick-lists [2] or SLAB [3]. Both these approaches try to keep low the volume of concurrent accesses to the core allocator that is in charge of ultimately delivering memory, either logical or physical. In the approach in point (a), this is achieved by resorting to the core allocator only when the thread's own pre-reserved arena, which is not accessed by other threads, gets exhausted. In the approach in point (b), cached allocation diminishes the pressure of concurrent accesses to the core allocator—e.g. a kernel-level buddy system—by having higher-level allocators destined to serve specific memory requests, such as those associated with a given size and/or memory alignment—this is the classical case of kernel-level page-table or SLAB allocations. In such a case, concurrent threads performing memory allocations undergo a coordination phase only when they need to access the same cached allocator instance or when this allocator is exhausted and a new memory segment needs to be taken from the core allocator.

Concurrent allocation/release operations have also been handled by creating data separation on the core allocator via multi-instance approaches, and redirecting the requests towards different instances. This further increases the likelihood of saving the requests from actual conflicts that may lead one or more of them to be delayed. For instance, this approach is used in OS-kernel physical memory management on large scale NUMA (Non-Uniform-Memory-Access) machines, where multiple disjoint instances of the buddy system are included, each one managing physical frames to be allocated from—or released to—different NUMA nodes. This has a crucial role to ensure that threads can easily retrieve memory from the NUMA node they reside in.

Generally speaking, we can frame our contribution by devising the notion of *back-end* and *front-end* allocators. The former, which are those we have previously referred to as core allocators, handle the lowest level of memory management. The latter are built on top of the back-end allocator with the goals of (i) reducing the access pressure to it and (ii) satisfying more specific purposes.

In this article, we tackle the issue of scalability of back-end memory allocation, which is an orthogonal approach with respect to reducing the request pressure on the back-end level by designing front-end allocators, e.g., adopting (a), (b), or multiple-instance approaches. Specifically, we present a *non-blocking back-end memory allocator* implement-

- *R. Marotta and M. Ianni are with the University of Rome La Sapienza. E-mail: {marotta,mianni}@diag.uniroma1.it*
- *A. Pellegrini is with Lockless S.r.l. E-mail: pellegrini@lockless.it*
- *F. Quaglia is with the University of Rome Tor Vergata. E-mail: francesco.quaglia@uniroma2.it*

Manuscript received XX; revised XX.

ing the buddy-system specification. We refer our solution to as NBBS (Non-Blocking Buddy System).

In our solution, concurrent allocations/releases are not coordinated via spin-locks. Rather, actual coordination, and the guarantee of an always-coherent state of the buddy system, are achieved by only relying on Read-Modify-Write (RMW) instructions executed along the critical path of allocation/release operations. These instructions are exploited to detect whether concurrent requests have conflicted on the same portion of the allocator metadata. This may lead a few requests to be aborted and retried, as in the classical non-blocking algorithmic paradigm devised in [4]. However, if conflicts do not materialize, then our proposal fully saves the latency that would be spent by lock-based approaches, which temporarily block concurrent operations a-priori of their execution.

Our non-blocking solution can be used in combination with any already existing scheme aimed at diminishing the pressure of concurrent accesses to the back-end allocator, e.g., by introducing multiple instances or combining it with front-end allocators. This is because our unique goal is to provide a memory allocation system that optimizes the management of concurrent accesses with respect to lock-based approaches. On the other hand, having a more efficient back-end allocator can allow to reduce the impact of, e.g., pre-allocation on actual memory unavailability in scenarios where there are skewed memory usages by different threads—so that the pre-reserved memory for a given thread cannot be used for serving a more memory-demanding one.

Our buddy-system implementation has been released as free software[1], and we also provide a proof of its correctness as well as experimental data demonstrating the actual scalability of our proposal.

The remainder of this article is structured as follows. In Section 2 we discuss related work. The non-blocking buddy system and its correctness proof are presented in Section 3 and Section 4, respectively. Experimental data are provided in Section 5.

2 Related Work

The theoretical foundations of non-blocking concurrent algorithms have been presented by the seminal work in [4]. This work has opened the path towards the design and implementation of algorithms that can fit well the scalability requirements imposed by modern (large-scale) multi-core machines. Also, the avoidance of lock usage in such new algorithmic class has indirectly offered the opportunity to develop coordination algorithms that are more suited for CPU-stealing contexts such as Cloud-based computing. In these contexts, the de-schedule of a lock-holding thread because of CPU-steals can lead to detrimental effects on performance—and waste of energy—because of the stretch of the spin-locking phase by other threads attempting to access the same critical section. Along this direction, a lot of effort has been spent in developing non-blocking versions of classical data structures such as lists or queues [5], [6], hash-tables [7], registers [8], [9] binary-search trees [10], [11] and priority queues [12], [13].

1. https://github.com/HPDCS/NBBS

Compared to all these solutions, our proposal is orthogonal since we focus on a buddy-system data structure. Also, while many solutions in the memory allocation context have been devised in order to reduce the negative impact of concurrency and synchronization on memory allocation/release by relying on pre-reserving or caching, no one fully faces the problem of concurrent accesses to back-end allocators, e.g., a common OS-level buddy system.

Hoard [14] has been a breakthrough in memory allocator design. This solution is based on pre-reserving memory to be delivered to specific threads (or CPU-cores), and resorts to lock-based coordination across the threads whenever the pre-reserved memory is fully used by a thread and the global state of the memory allocator needs to be changed in order to provide a new pre-reserved area. Similar approaches, where threads operate on pre-partitioned heaps (hence on different memory-allocator instances), have been later presented. The work in [15] provides non-blocking capabilities of memory allocations "with high probability", just depending on the pattern of memory allocations/releases and the overall memory usage. To this goal, it exploits the concept of Restartable Critical Section, which is an invasive approach to provide non-blocking progress by interacting with the underlying operating system. Finally, this proposal does not address the problem of avoiding blocking allocations/releases in scenarios where a same allocator instance can be concurrently accessed by multiple threads. After that, several works were able to provide lock-free adaptations of the original Hoard allocator. To the best of our knowledge, the first one to achieve this goal is the lock-free allocator described in [16]. Later, the authors of NBMalloc [17] designed a new data structure, called *flat-set*, to provide non-blocking management of pre-allocated blocks within per-processor heaps, reducing the probability of false sharing and external fragmentation. Similarly, SF-Malloc [18] uses a non-blocking stack data structure to post memory across different threads within the pre-reserving scheme. Finally, the authors of LRMalloc [19] were able to further improve the performance of their predecessors by introducing an optimized per-thread cache.

The solution in [20] is suited for SIMD systems and exploits worker threads operating in isolation (on different data portions) in order to deliver memory to a pool of requesting threads. The workers do not block each other thanks to pre-partitioned accesses to the allocator data structures—they actually operate on different allocators. Hence this approach does not provide a mechanism to perform non-blocking memory allocation/release on a same shared instance of the allocator, which is the target of our buddy system. On the other hand, this solution looks to be a reasonably scalable approach for devices such as GPUs, where multiple threads typically do the same kind of operations in parallel, such as requesting memory to the worker in charge of managing the allocator, which ultimately delivers a unique memory block, each portion of which is used by a different requesting thread.

Our approach is de-facto orthogonal to any approach that tries to improve the scalability of memory allocation via pre-reserving, since we optimize the actual handling of concurrent operations on the core allocator—a buddy system in our case—on top of which pre-reserving can be

put in place. Furthermore, it allows to cope with scenarios where pre-reserving or cached allocation could be not fully adequate. In particular, cached allocators and multi-instance approaches do not fully cope with skews in memory utilization by different caches/instances. This is the case of OS-kernel physical memory allocation, where requests can be issued by active threads either for the execution of specific system calls, or because of the materialization of logical pages into physical memory upon user-space code accesses to a previously mapped logical memory region—like an `mmap`-ed page on Unix systems. In these scenarios, the skew of memory requests towards a given instance of allocation service—such as an allocator operating in a given NUMA node selected on the basis of memory-policies associated with the requesting threads—can give rise to a peak of requests saturating cached allocation and requiring coordinated concurrent accesses to the underlying buddy-system instance.

A solution with goals similar to the ones pointed out by our proposal can be found in [21]. Here the authors present a concurrent non-blocking memory allocator relying on a kind of *conditional atomic dec* instruction, which is not currently supported by conventional processors. Conversely, our proposal only relies on machine instructions offered by off-the-shelf CPUs. Moreover, as pointed to by the same authors, the solution in [21] is not able to effectively detect fragmentation, thus possibly leading to a large amount of false negatives (allocation failures) due to fated-to-fail allocation attempts that cannot be satisfied via contiguous and correctly aligned memory. This problem is avoided in our solution since memory allocations are guaranteed to attempt the acquisition of memory chunks that are always compliant with the request (for both size and memory alignment).

The work in [22] presentes an allocator which provides wait-free constant-time allocations and releases. However, this work has several limitations that cannot be fairly compared with our proposal. In more detail, its memory operations target fixed-sized blocks and an a-priori known number of threads.

Finally, COA [23] provides a scheme to fragment and coalesce arbitrary-sized memory blocks in a lock-free manner. Moreover, in order to provide no constraints to the block sizes, it needs additional dynamic non-blocking data structures (i.e. a lock-free binary tree [11]), and, hence, it requires dynamic allocation of items (nodes) and a memory reclamation scheme (e.g. epochs [24] or hazard pointers [25]). Consequently, this work does not implement the buddy-system specification and it is not best suited to be the lowest-level allocator in the memory-management architecture.

3 NBBS

3.1 Basics: Buddy-System Specification

A buddy system divides a contiguous memory region into partitions, namely memory blocks, by recursively splitting it into halves. These partitions are always contiguous, thus minimizing external fragmentation. In order to handle a memory allocation request, it looks for a memory block large enough to satisfy the request. Every memory block is associated with an *order*, namely an integer ranging from 0 to a specified upper limit. The size of a block of order n is proportional to 2^n, so it is exactly twice the size of blocks that are at one order lower.

The peculiarity of a buddy system is that memory allocations can be satisfied by: (1) returning a block if already available; (2) splitting higher-order free blocks into smaller ones; (3) merging lower-order free blocks into a larger one.

TABLE 1
Bitmasks used to manage the status bits of a node.

SYMBOL	VALUE
`OCC_RIGHT`	`0x1`
`OCC_LEFT`	`0x2`
`COAL_RIGHT`	`0x4`
`COAL_LEFT`	`0x8`
`OCC`	`0x10`
`BUSY`	(`OCC` | `OCC_LEFT` | `OCC_RIGHT`)

3.2 NBBS Memory Layout

Our non-blocking buddy system keeps track of the state of the memory segment used for serving allocations by the means of a static complete binary tree. The tree has a predefined maximum depth D. The root of the tree corresponds to (and keeps track of the state of) the entire memory segment used to serve allocations. Each child of a node represents a portion (one half) of the parent's chunk of memory, while the leaves represent the state of the minimum allocable memory chunks, called *allocation units*. All the nodes with depth i belong to the i-th level of the tree[2] and they are associated with the order $D-i$ within the buddy system specification. In particular, according to the classical buddy-system structure, if a node at level i has size s, the children of this node, located at level $i+1$, have size $s/2$, and the union of the blocks of memory associated with the children forms a larger block of memory that exactly corresponds to the parent. The overall size of memory managed by the buddy system is equal to `total_memory`. Then, the memory size managed by a node at level i is equal to $\texttt{total\_memory}/2^i$, and the allocation units (corresponding to the leaves of the tree) have size equal to $\texttt{total\_memory}/2^D$.

In our non-blocking buddy system, each node in the tree embeds a bitmap with 5 bits. They are used to represent the state of the node itself—thus of the corresponding memory chunk—and of its sub-trees (if any) according to the following semantic: i) `occupied` indicates whether an allocation operation has targeted exactly that node, meaning that an allocation request has been served by the memory chunk corresponding to that node; ii) `left_occupied` and `right_occupied` signal if the branches (left and right, respectively) covered by the node are totally or partially occupied. This indicates that some allocation request has been served by a node in these sub-trees; iii) `left_coalescent` and `right_coalescent` indicate whether a memory release operation is currently in place in any of the two sub-trees. In other words, these two flags indicate whether the node is currently in a transient state because of some memory release running on a sub-tree. The bitmasks shown in Table 1 are used to extract and manipulate the status bits associated

2. Usually, the level of a node is defined to be equal to its depth plus 1. We prefer the convention adopted in [26], where the level counting starts from 0. This allows us to maintain the concept of level and depth aligned (no by 1 offset).

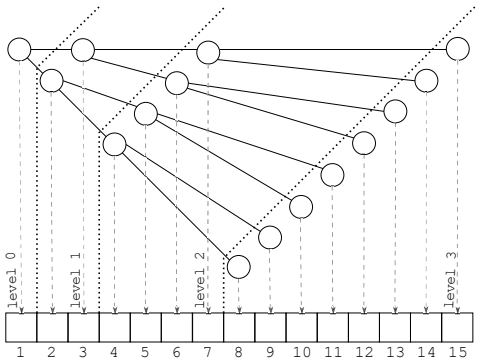


Fig. 1. Array representation of the tree structure.

Algorithm 1 Helping functions to manipulate bitmaps in the non-blocking buddy system.

procedure CLEAN_COAL(int val, int $child$)
  return val & $\neg$(COAL_LEFT >> $\text{mod}_2(child)$)
procedure MARK(int val, int $child$)
  return val | (OCC_LEFT >> $\text{mod}_2(child)$)
procedure UNMARK(int val, int $child$)
  return val & $\neg$((OCC_LEFT|COAL_LEFT) >> $\text{mod}_2(child)$)
procedure IS_COAL(int val, int $child$)
  return val & (COAL_LEFT >> $\text{mod}_2(child)$)
procedure IS_OCC_BUDDY(int val, int $child$)
  return val & (OCC_RIGHT << $\text{mod}_2(child)$)
procedure IS_COAL_BUDDY(int val, int $child$)
  return val & (COAL_RIGHT << $\text{mod}_2(child)$)
procedure IS_FREE(val)
  return $\neg(val$&BUSY$)$

with the nodes of the tree. In order to correctly manipulate the status bits while handling concurrent operations, our solution relies on RMW instructions offered by conventional architectures, like x86. In particular, our operations are based on the Fetch&Add (FAD) and Compare&Swap (CAS) RMW instructions. In this work, we use two variants of the latter: one returns the success (or the failure) of the operation (denoted as BCAS), while the second provides the value of the memory location before its execution (VCAS).

Each memory allocation request of a given size s is mapped to the lowest-order chunk capable of satisfying it. Also, when a node is allocated, the sub-tree having the node as its root is considered to be fully occupied—this is exactly what the occupied flag takes care of signaling. Nevertheless, we do not reflect node occupancy on its descending nodes, which helps saving RMW instructions for updating the corresponding status bits. This could create the illusion that a node is allocable while being actually not, due to a previous—or concurrent—memory request bound to a larger size (i.e. to an ancestor node in the tree). This scenario is anyhow correctly handled by our solution.

We represent the tree keeping track of the buddy-system state with an array of $2^{D+1} - 1$ elements, which we refer to as tree[]. We place the root node at index 1 and exploit the conventional rule that, given a node with index n, the left child of this node is at index $2n$ and the right child is at index $2n + 1$. This representation fulfills the condition that nodes belonging to the same level of the tree are placed in a contiguous portion of the array, thus simplifying the search for free chunks of a given size while performing allocations. In fact, starting from the amount of requested memory s it is possible to compute the target level—the one containing nodes useful to serve the request based on its size—as $level = \lfloor log_2(\texttt{total\_memory}/s) \rfloor$ upper bounded by the value D. The nodes belonging to this level are those with index $n \in [2^{level}, 2^{level+1} - 1]$. A graphical representation of this structure is shown in Figure 1. Denoting with base_address the start of the memory region (either physical or logical) managed by the allocator, for each node with index n, it is possible to compute the starting address $\texttt{starting}_\texttt{n}$, the size $\texttt{size}_\texttt{n}$ of the corresponding memory chunk and the level associated with the node, according to the following rules:

$$\texttt{level}_\texttt{n} = \lfloor log_2(\texttt{n}) \rfloor \quad (1)$$

$$\texttt{size}_\texttt{n} = \frac{\texttt{total\_memory}}{2^{\texttt{level}_\texttt{n}}} \quad (2)$$

$$\texttt{starting}_\texttt{n} = \texttt{base\_address} + (\texttt{n} - 2^{\texttt{level}_\texttt{n}}) * \texttt{size}_\texttt{n} \quad (3)$$

Finally, we couple the tree[] array with another array, called index[]. This array is used to keep track of the indices of the nodes that have been used for serving memory requests, which have not yet been released. Given the address, we can easily retrieve the index of the node and use this information during a release operation, whose API receives the base address of the to-be-freed node as its unique parameter.

We describe our non-blocking buddy system by relying on a few additional notations. The deepest reachable level in the tree-based organization is stored in a variable denoted as depth. Moreover, we denote as min_size the variable keeping the size of the allocation units associated with the leaves, while max_size maintains the maximum amount of memory allocable with a single request—clearly, $\texttt{max\_size} \leq \texttt{total\_memory}$. Nodes, whose corresponding size is equal to max_size, are all located at the same specific level, denoted as base_level, which might not be the same one of the root—i.e. the maximum allocable size is smaller than the whole available memory. Consequently, each memory allocation/release operation consists in traversing the tree up to base_level in order to correctly manipulate the status bits of the traversed nodes. In fact, these need to be (re)aligned to the new state of the buddy system, depending on the type of operation being performed. In particular, a memory release operation starts from the node to be released, while a memory allocation operation starts from whichever node at the target level—that depends on the size of the memory allocation request.

Given a value val of the status bits of a given node and the index $child$ of the previous node traversed while moving towards the root, which is a child of the given node, Algorithm 1 shows the status-bits manipulation functions we used. CLEAN_COAL$(val, child)$ clears the coalescing bit relative to the branch of the child. MARK$(val, child)$ sets the occupancy bit of the branch of the child. UNMARK$(val, child)$ clears both the coalescing and the occupancy bits relative to the branch of the child. IS_COAL$(val, child)$ returns *true* if the coalescing bit relative to the child is set. IS_OCC_BUDDY$(val, child)$ returns true if the occupancy bit relative to the buddy associated with $child$ is set.

Algorithm 2 Allocation - Part A

```
A1:  procedure NBALLOC(size_t size) return void *
A2:      if size > max_size then
A3:          return NULL
A4:      ts ← release_count
A5:      level ← min(⌊log2(total_memory/size)⌋, depth)
A6:      for i ← 2^(level−1) to 2^level − 1 do
A7:          if IS_FREE(tree[i]) then
A8:              failed_at ← TRYALLOC(i)
A9:              if ¬failed_at then
A10:                 index[(starting_i − base_address)/min_size] ← i
A11:                 return starting_i
A12:             else
A13:                 d ← (1<<(level_i − level_failed_at))
A14:                 i ← (failed_at + 1) · d
A15:     if ts ≠ release_count then
A16:         goto A4
A17:     return NULL
```

IS_COAL_BUDDY$(val, child)$ returns true if the coalescing bit relative to the buddy associated with $child$ is set. In order to capture whether a node is a right or left child, all these functions use a two-modulus operation applied to its index. Actually, we have implemented $\bmod 2$ operations in a bitwise fashion by simply checking the value of the least significant bit. Finally, we additionally use IS_FREE(val) to detect whether a node, whose status bits are embedded in the val bitmap, is currently free. This condition is verified when the node itself has not been reserved for some allocation operation (it is not occupied), and none of its left and right sub-trees has nodes currently reserved for allocations—none of them is partially or fully occupied.

Similarly to most common allocators, our non-blocking buddy system exposes two API functions for either requesting a chunk of memory of (at least) a given size or for releasing some previously allocated memory chunk identified by its address. Both `tree`[] and `index`[] are initialized to zero at start-up. Recall that index 0 does not correspond to any node of the tree since the initial element of `tree`[] is associated with index equal to 1. Setting the entries of `index`[] to zero indicates that none of the memory chunks (and none of the corresponding addresses) managed by the buddy system at any level has been delivered for usage.

For the sake of simplicity, the allocation/release algorithms discussed in the following sections assume a sequential consistent memory model in the underlying hardware. A discussion about implementing such algorithms on more relaxed memory models is conducted in Section 3.5.

3.3 Memory Allocation Algorithm

The non-blocking memory allocation operation is divided in two algorithms, NBALLOC() and TRYALLOC(). The pseudocode of NBALLOC(), which represents the memory allocation API actually exposed to the user, is reported in Algorithm 2. NBALLOC() takes the size of the memory allocation request as input and, if such amount of memory is available, it returns the address of a memory chunk large enough to fit the request. If the size exceeds the overall memory allocable by a single invocation, the allocation fails. Conversely, if it is smaller than the minimum amount managed by the buddy system, it is rounded to the allocation unit, namely the size associated with the leaves. In the general case, i.e. the request size is legitimate, the target level of nodes to be considered for allocation is obtained by

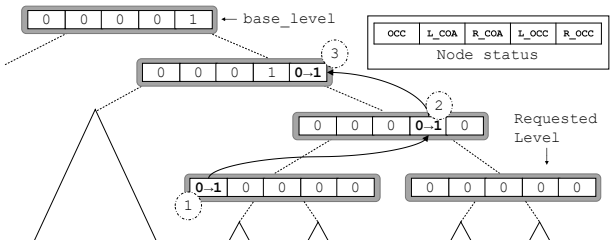


Fig. 2. Visual representation of TRYALLOC operations.

Rule 1 (line A5). Once identified the right level, thus the range of indices of nodes suitable for the allocation, these nodes are scanned in order to search for a free one. Note that, differently from what is shown in Algorithm 2, not necessarily such a search has to start from the first node at that level. For instance, starting from scattered points will more likely lead concurrent allocations (bound to that same level) to target different free nodes, if any. When a free node is found (line A7), the allocation operation tries to: i) acquire it by setting its status as occupied; ii) fragment its ancestors by setting their status as left/right occupied. The latter phase can complete when there is no need to fragment nodes, i.e. we reach the `base_level`, namely the one containing nodes corresponding to the maximum size allowed for memory requests. However, if some ancestor is already fully occupied, we cannot fragment it. Thus, the allocation attempt fails and we restart from the end of the memory region that was detected as fully occupied. All this steps are included within the TRYALLOC() procedure. This returns zero upon success. Otherwise, it returns the index of the node that makes the allocation fail. In the latter case, the algorithm moves to the next candidate node by exploiting the index returned by TRYALLOC() (lines A13-A14) to skip the whole sub-tree relative to the node causing the failure. If no node at that level is found to be free, the allocation operation fails if no concurrent release has completed in the meanwhile, indicating that the current state of the buddy system is not compliant with the issued request. Otherwise, if some release operation has been concurrently completed, the algorithm starts again searching for a free node. This can be easily detected by checking if a counter incremented upon each release completion has been updated concurrently.

When the TRYALLOC() procedure succeeds at reserving a node, the relative address with respect to `base_address` is computed, and the corresponding entry of the `index`[] array is updated to store the index of the reserved node (line A10). This allows us to retrieve the level of a node targeted by a release operation in constant time. Then, the memory address is returned, indicating a successful allocation.

The pseudo-code of TRYALLOC() is reported in Algorithm 3 and a visual representation of its steps is shown in Figure 2. It takes the index of a node (previously observed as free) as input and tries to (i) occupy this node, and (ii) propagate the information about the occupancy up to the ancestor node belonging to `base_level`. The former task is carried out by relying on the `CAS` machine instruction (see line T2 corresponding to step 1 in Figure 2), which atomically tries to update all the occupancy bits in the status

Algorithm 3 Allocation - Part B

```
T1:  procedure TRYALLOC(index n) return index
T2:      if ¬ BCAS(&tree[n], 0, BUSY) then
T3:          return n
T4:      current ← n
T5:      while level_current > base_level do
T6:          child ← current
T7:          current ← current>>1
T8:          do
T9:              curr_val ← tree[current]
T10:             if curr_val & OCC then
T11:                 FREENODE(n, level_child)
T12:                 return current
T13:             new_val ← CLEAN_COAL(curr_val, child)
T14:             new_val ← MARK(new_val, child)
T15:         while ¬ BCAS(&tree[current], curr_val, new_val)
T16:     return 0
```

bitmap of that node and to check if the status bits are still set to zero. If the CAS fails, it means that something has concurrently changed on the state of the node, affecting the ongoing operation. Therefore, the memory allocation operation needs to be aborted and retried on a different node just as in the spirit of non-blocking coordination algorithms. Conversely, if the CAS succeeds, the procedure continues by traversing the nodes along the path from the one currently occupied towards base_level. This traversal is required to update the left/right occupancy bits of ancestor nodes to reflect that their sub-tree has become partially occupied. In this way the corresponding memory chunk will figure out as fragmented into lower-order chunks, preventing other allocations from occupying an ancestor of the just allocated node. For each node along the path towards base_level, the procedure tries to mark its state as left or right occupied, depending on what branch we have traversed (e.g., steps 2 and 3 in Figure 2). These operations are still performed via the CAS machine instruction (see line T15) with the peculiarity that, if it fails, it can be retried. In fact, the update we are trying to perform can be still coherent with respect to the memory allocation we are carrying out and other concurrent operations. For instance, a CAS on the node we are traversing may fail since a concurrent operation updates the occupancy bit associated with the other branch of the tree, or even the same branch. The scenario where a failure of the CAS at line T15 indicates that the operation needs to be aborted—hence it is not compatible with any other concurrent operation—only occurs when another operation updates the (fully) occupied bit. In this case, another operation has reserved exactly that node—and the corresponding memory chunk—for a concurrent (or already-finalized) memory allocation. Hence, we cannot fragment that chunk to reserve some buffer at a lower-order. We also note that, while attempting to set to 1 the left/right occupancy bit of a node along the traversal, the corresponding left/right coalescing bit is contextually set to 0. As we will clarify in the next section, this is required to make conflicting releases aware that the branch is involved in a new allocation and cannot be freed.

If the base_level node is reached along the backward traversal, the node originally targeted when starting TRYALLOC() can be considered as correctly taken. Conversely, if some node with the occupied bit equal to 1 is found along the path, the current allocation fails and nodes "speculatively" updated along the backward traversal have to be restored to their original value. This is done by invoking the FREENODE() procedure in Algorithm 4, which is also used to support non-blocking memory release operations in our buddy system. As pointed out, in such a scenario the TRYALLOC() procedure ends returning the index of the node for which the allocation has failed.

3.4 Memory Release Algorithm

A memory release is composed of three phases. In the first phase, the ancestors of the node to be released are marked as coalescing in order to notify that a release operation is in place along the corresponding path towards the root. In the next phase, the node to be released is marked as free by resetting all its occupancy bits. During the last phase, all the nodes previously marked as coalescing are updated again to notify that the sub-tree involving the just released node is actually free—therefore it can serve again memory requests. The first two steps are implemented by the FREENODE() procedure (see Algorithm 4), while the last one is carried out by the UNMARK() procedure (see Algorithm 5).

The FREENODE() procedure is not directly exposed to the user. It is encapsulated in the NBFREE() procedure, which is the actual memory release API in our non-blocking buddy system. This procedure receives the memory address corresponding to the chunk to be released, computes the relative index of the corresponding node in constant time by inspecting the index[] array, and triggers the execution of FREENODE().

FREENODE() takes as input the index of the node to be released and an *upper bound*, which identifies the upper level to be reached along the backward traversal associated with its execution. If the FREENODE() procedure is invoked by NBFREE(), the upper bound is set to base_level, making it traverse the tree up to the level corresponding to the maximum allocable size. In this case, we are releasing a previously allocated node and hence the status bits need to be reflected up to the maximum useful level of the tree. Conversely, if the procedure is invoked by a failed TRYALLOC() (see line T11), the upper bound is set to the level of the last updated node during an aborted memory allocation. As discussed before, this execution path is related to the need for TRYALLOC() to restore the status of nodes involved in the traversal towards the maximum level, which was interrupted because it found an already occupied node.

While traversing all the nodes up to the upper bound, FREENODE() atomically sets the coalescing bit of the correct position (value 1), via the CAS machine instruction (see line F13 in Algorithm 4). A visualization of this phase is shown in Figure 3. If along this path a buddy is detected as fragmented by other allocations, the climb is stopped early, since the corresponding sub-tree cannot be considered as free. In fact, nodes at upper levels in the tree will be still fragmented regardless the completion of the on-going release, because there is some allocated node in the opposite branch—this is the case shown in Figure 3, where the right sub-tree of the node below the upper bound is already fragmented.

Once the first phase is concluded, FREENODE() can start signaling that the involved node has been released. These steps are sketched in Figure 4. In particular, the node to be released can be updated by resetting its occupancy bits with a simple write. This takes place by simply writing zero on all

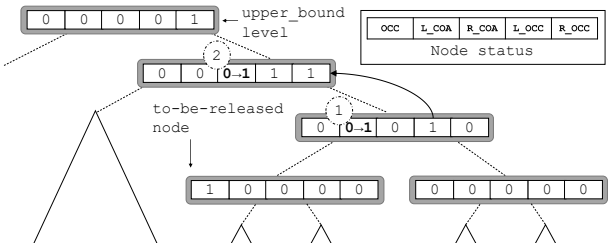


Fig. 3. First phase of the FREENODE (lines F6-F19 of Algorithm 4).

Algorithm 4 Memory Release

```
F1:  procedure NBFREE(void *addr)
F2:      n ← index[(addr − base_address) / min_size]
F3:      FREENODE(n,base_level)
F4:      FAD(&release_count,1)
F5:  procedure FREENODE(index n, index upper_bound)
F6:      current ← n>>1
F7:      child ← n
F8:      while level_child > upper_bound do
F9:          or_val ← MARK(0,child)
F10:         do
F11:             curr_val ← tree[current]
F12:             new_val ← curr_val | or_val
F13:             old_val ← VCAS(&tree[current],curr_val,new_val)
F14:         while old_val ≠ curr_val
F15:         if IS_OCC_BUDDY(old_val, child) ∧
F16:                                 ¬IS_COAL_BUDDY(old_val, child) then
F17:             break
F18:         child ← current
F19:         current ← current>>1
F20:     tree[n] ← 0
F21:     if level_n ≠ upper_bound then
F22:         UNMARK(n,upper_bound)
```

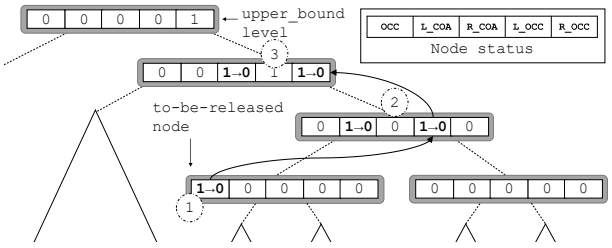


Fig. 4. Second (line F20 of Algorithm 4) and third (UNMARK given in Algorithm 5) phase of the FREENODE (Algorithm 4).

the status bits (see line F20 in Algorithm 4, which matches step 1 of Figure 4).

The last phase is responsible for propagating the node release up to the upper bound and, possibly, merging buddies. This is achieved by invoking UNMARK(), which traverses the nodes from the one to be released towards the upper bound, cleaning the left/right coalescing and occupancy bits of the traversed nodes (see steps 2 and 3 in Figure 4). For each node met, the first step is to verify whether the coalescing bit is still set (see lines U8 and U12): if it is not, the procedure returns, finishing the release operation. As hinted before, this scenario is possible if some allocation/release in the same sub-tree of the target node has already occupied/released that coalescing sub-tree. In fact, allocations clean coalescing bits to deal with this scenario.

Conversely, if the coalescing bit is still set, the procedure tries to clean both the coalescing and the occupancy bits atomically (line U11). This operation is done via CAS in a retry-cycle in order to manage the case in which the coalescing bit has been reset by some concurrent operation

Algorithm 5 Unmark

```
U1:  procedure UNMARK(index n, index upper_bound)
U2:      current ← n
U3:      do
U4:          child ← current
U5:          current ← current>>1
U6:          do
U7:              curr_val ← tree[current]
U8:              if ¬IS_COAL(curr_val, child) then
U9:                  return
U10:             new_val ← UNMARK(curr_val, child)
U11:         while ¬ BCAS(&tree[current],curr_val, new_val)
U12:     while (level_current > upper_bound) ∧
U13:                                 ¬IS_OCC_BUDDY(new_val, child)
```

(the resource has already been reused/released). Then, the procedure checks if the buddy occupancy bit is set at each step and, in the positive case, it returns. In fact, similarly to the first phase, if such buddy is occupied, we cannot propagate the node release and merge buddies in the upper levels of the tree, since the chunks associated with these buddies are still fragmented.

Overall, beyond providing non-blocking capabilities while allocating or releasing memory at a given level, our buddy system allows fragmenting and merging operations—which logically move nodes across different levels within the allocation scheme—still in non-blocking fashion. Usually, these operations have to be performed explicitly, but in our approach they are carried out implicitly while performing allocations and releases.

3.5 Implementation with Relaxed Memory Models

Most of the updates of the buddy-system metadata, concurrently accessible by threads, are performed in our algorithms with atomic RMW instructions, that are totally ordered with all other memory accesses. However, there are two write operations that need special care when deploying the algorithms on machines implementing a relaxed memory-consistency model (like machines with non-FIFO store buffers). The first one is at line A10 of Algorithm 2, where the procedure writes the index array. The cells of this array are read during release operations for speeding up the retrieval of the node to be freed. In this case adding a memory barrier is enough for ensuring that another processor gets the correct value to be used in a subsequent memory release. The second write is the one for signaling the release of a node previously occupied by an allocation (see line F20 of Algorithm 4). Since the status of a node is read before attempting to acquire it, processors might not detect that a release has completed. Also in this case, a simple memory barrier is enough to avoid this issue, but it is suboptimal. In fact, we can exploit the full barrier already materialized by the atomic instructions of the UNMARK procedure used to merge buddies (line U11 of Algorithm 5) and use an explicit memory barrier in the release if and only if the climb towards the root stops at the very first step—in such a scenario the UNMARK procedure does not execute any RMW instruction.

3.6 Optimizations

3.6.1 Free-Slot Search

On the one hand, the allocation algorithm performs a logarithmic number of steps for updating the tree. On the

other hand, it performs a linear search to find a free slot for executing the TRYALLOC. This is because the nodes' state does not provide information about the amount of free memory available in its sub-trees. Consequently, we cannot exploit binary search for finding a free node, e.g., by descending from the root. For this reason, we have devised an optimization inspired to the allocation of file descriptors in the Linux kernel. In particular, for each thread we maintain (using per-thread memory) a pointer to the last freed node (by that same thread) for each level. So we essentially cache some node reference on a per-thread basis. Consequently, a thread first tries to allocate the node with cached reference and then, if the allocation attempt fails, it starts a linear search for a free slot. Finally, the cached information is updated in a successful allocation by making it point to the next node at the same level. Here, the idea is that the next node is expected to be available when the next allocation for the same size will occur.

Such an approach is known to provide constant-time search in practice and, thus, it is embedded by default in our non-blocking buddy-system implementation. As a last note, introducing this "software cache" is not equivalent to applying pre-reserving by introducing per-thread heaps. In fact, nodes with references stored in a cache of a thread are still available for allocations made by other threads. On the other hand, initially populating the caches of the different threads with random references to nodes allows us to reduce the likelihood of actual conflicts in the allocation.

3.6.2 4-Level Packaging

In our solution, a thread performing successful memory allocations/releases executes (at worst) a number of RMW instructions equal to the depth of the acquired/released node with respect to the `base_level` value within the tree. However, reducing the number of RMW instructions executed along the critical path of any thread in a non-blocking algorithm is an important aspect in terms of further performance improvements. To this goal, we packed the state of a bunch of nodes, namely a node and some of its descendants, in the same status bitmap by exploiting the fact that the number of status bits required to represent the state of each node is significantly smaller than the word size—which is 64-bits on nowadays conventional machines. With this organization a thread is able to update at one time the state of multiple levels with a single RMW atomic instruction. Additionally, the coherence with the original specification is guaranteed by the fact that all state-update operations are performed via `CAS`, which succeeds only if the status bit-mask has not changed (in any of its bits) in the meanwhile. We note that, beyond the possibility to improve performance depending on the specific workload, we also have the advantage of reducing the actual amount of memory locations required for storing the non-blocking buddy-system metadata. On the down side, a compact representation of the tree increases the likelihood of conflicts across concurrent threads.

In more detail, given a generic node in the tree, its state can be derived by looking at the state of its children, as shown in Figure 5. In fact, the partial occupancy of a node—say its left/right occupancy—can be computed with a logical OR operation on the (partial and full) occupancy

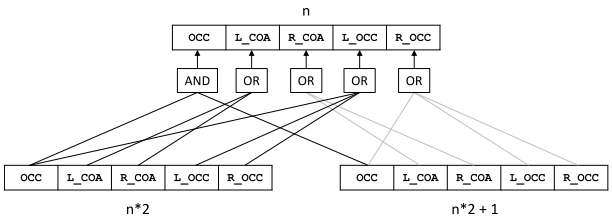


Fig. 5. Derivation of node state by children' state.

bits of the children nodes. In the same way, the coalescing bits can be derived from the ones of the children. Moreover, when a node is actually set as `occupied` also its children can be logically considered to reside in the same state. In fact, they cannot be considered as freely available for allocation. In the same way, occupying two buddies looks to be the same as fully occupying the parent node. This means that, similarly to partial occupancy, full occupancy of a node can be computed with a logical AND operation on the `occupied` bits of its children.

Such a reasoning can be recursively applied to all the ancestors of a node, hence a sub-tree starting from a given node can be represented by the nodes in underlying levels. Then, considering a word size equal to w bits, and a status bitmap of size s bits, we can pack in a single w-bit variable the states of a bunch of nodes of depth equal to d, with $2^d \cdot s < w$, thus representing the state of $2^{d+1} - 1$ nodes of the original tree. In our case, we are able to manage 4 levels, namely 15 nodes with 8 nodes (40 bits) in the fourth level, in a single 64-bit word, reducing the number of atomic RMW operations by a factor of 4, and the memory footprint by a factor of 15 with respect to the original implementation. A representation of this transformation is shown in Figure 6 and its impact on performance will be evaluated and discussed in Section 5.

Of course, this means that, when an operation is performed on a node that is not a leaf of a bunch, its state has to be decoupled from its descendant in the same bunch. To practically implement the operation in this variant of our data structure, each node in the `tree[]` array stores a pointer to its bunch and its position inside it (computed the same way as in the original solution). The algorithms for the 4-level solution are pretty similar to those shown before, with two main differences. First, the direct allocation/release of a node has to check and then set the state of all the nodes in the sub-tree in the deepest level of the bunch. In particular, given a node n with depth d_n, its level D_n and position b_n inside the bunch can be computed as:

$$D_n = d_n \mod 4$$
$$b_n = n \mod 2^{D_n} + 2^{D_n}$$

Then, the corresponding bunch-leaf nodes have position p in the bunch equal to:

$$p \in [b_n \cdot 2^{3-D_n}, (b_n + 1) \cdot 2^{3-D_n} - 1]$$

Second, updates of the bunches up to the root have to be executed with a step of 4 levels in order to update each time the relative bunch-node.

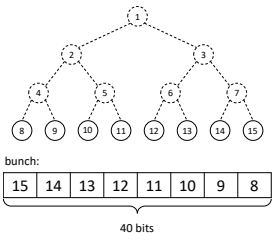


Fig. 6. Array representation of a bunch.

4 Progess and Safety properties

4.1 Progress

In this section, we provide a proof that our buddy system achieves *lock freedom* [27], thus guaranteeing that *some* allocation/release invocation eventually completes. Informally, it ensures that if a thread does not make progress (i.e. its operation aborts and needs to be retried indefinitely) this is *due to* the advancement (the progress) of other threads. Our proof proceeds in two steps. First, we show that both allocation and release operations are lock-free if the procedures they rely on are lock-free. Then, we prove that threads executing those procedures, which are stuck in a retry loop, conflict with other threads which are anyhow making progress. Hence these procedures are lock-free.

Lemma 4.1. NBALLOC *is lock-free if* TRYALLOC *is lock-free.*

Proof. There are two nested loops in NBALLOC. The outer loop repeats if the inner loop ends without returning and the variable `release_count` has been updated in the meanwhile. Suppose by contradiction that NBALLOC is not lock-free because of an infinite repetition of the outer loop. In this case, concurrent release operations have completed atomically incrementing the `release_count` variable (see line F4), so they made progress. Therefore, the assumption is contradicted.

Consider now the inner loop. It is a finite loop, unless the number of different memory chunks associated with different nodes in the tree that needs to be traversed at the target level for the allocation is infinite. But this is impossible since it would mean that the buddy system would manage an infinite amount of memory. At each loop iteration, local values are computed and two functions are invoked: IS_FREE, which executes a fixed number of steps, and TRYALLOC. Therefore, being the inner loop of NBALLOC finite, NBALLOC completes if TRYALLOC completes. Hence, the claim follows. □

Lemma 4.2. NBFREE *is lock-free if* FREENODE *is lock-free.*

Proof. NBFREE entails two non-iterative statements and the invocation of FREENODE. Therefore, it completes, if FREENODE completes. Hence, the claim follows. □

By the structure of TRYALLOC, FREENODE and UNMARK, we can see that they contain one occurrence of two nested loops. These loops have a well-defined structure. The outer loop performs a *climb* from a source to a destination, while the inner loop is a retry cycle for updating node metadata with a `CAS`. Also, even though FREENODE invokes UNMARK, this does not happen in any cycle of FREENODE. Therefore there is no nested retry cycle in the code structure.

Lemma 4.3. *At least one instance of a retry cycle in* TRYALLOC, FREENODE *or* UNMARK *is guaranteed to make progress at any time under concurrent invocations of these functions.*

Proof. Suppose by contradiction that no retry cycle makes progress under concurrent invocations of these functions. For this to occur, we need that all the concurrently executed `CAS` instructions in TRYALLOC, FREENODE and UNMARK fail. But this is impossible since the processor firmware guarantees that at least one, among all the concurrent `CAS` successfully terminates, even in the scenario where the same memory location is targeted by the concurrent `CAS` instances. Hence, the assumption is contradicted and the claim follows. □

Lemma 4.4. TRYALLOC, UNMARK, FREENODE *are lock-free functions.*

Proof. These functions perform climbs and by Lemma 4.3 an individual step of the climb at some level L in the tree (associated with a retry cycle) is guaranteed to make progress in some of these functions under concurrent invocations. Unless the number of climb steps to be performed by these function invocations is infinite (i.e. it should span an infinite number of levels), which is impossible given the finite amount of managed memory, eventually some climb terminates. Hence, these functions are lock-free. □

Theorem 4.1. NBALLOC *and* NBFREE *are lock-free functions.*

Proof. The claim follows from Lemmas 4.1, 4.2 and 4.4 □

4.2 Safety

Safety is commonly intended as "nothing bad happens" with a specific (concurrent) algorithm. For a buddy system, safety properties can be expressed as: S1) A successful allocation returns a memory chunk, whose size is coherent with respect to the size of the allocation request. S2) A release request releases exactly the memory area targeted by the request (not a different one). S3) A successful allocation returns a memory chunk not already allocated (either partially or fully). S4) If there is enough memory available, an allocation request that executes in isolation succeeds.

S1 is needed to show that an allocation reserves the right amount of memory. S2 defines the behavior of an allocator in face of legal release requests. S3 is needed to ensure that memory is correctly acquired in face of multiple and concurrent requests. Finally, S4 is required to rule out trivial allocators that always provide no memory at all upon memory allocation requests.

We prove that S1-S4 hold, additionally showing that our solution actually coalesces memory upon release operations.

Lemma 4.5. *A successful* NBALLOC *returns contiguous memory addresses.*

Proof. A successfully completed NBALLOC returns an address $\texttt{starting}_i$ where i is the index of the node n_i related to the allocation within the tree. By definition, n_i manages $\texttt{size}_i$ addresses (Rule 2 in Section 3.2) from (and including) $\texttt{starting}_i$ (Rule 2). Hence, the claim follows. □

Lemma 4.6. *A* NBALLOC *allocating a chunk of memory at level L (in the tree-like structure) returns a base address B aligned with* $\texttt{min\_size} \cdot 2^{depth-L}$.

Proof. Let $B = \texttt{starting}_i$ be the base address returned by an allocation, i be the corresponding node within the tree and $L = \texttt{level}_i$ be the target level. Without loss of generality, we assume that $\texttt{base\_address} = 0$. Consequently, $B = (i - 2^L) \cdot \texttt{total\_memory} \cdot 2^{-L}$ by Rules 2 and 3. By definition, we know that there are 2^{depth} leaves within the tree, whose size is equal to `min_size`. It follows that $B = (i - 2^L) \cdot \texttt{min\_size} \cdot 2^{depth} \cdot 2^{-L}$. Hence, the claim follows. □

Lemma 4.7. *Each function execution, that has reached a node n during a climb, has successfully updated the state of all the traversed nodes up to n.*

Proof. A function that reaches a node n at a level l during a climb has performed a climb step from level $l - 1$. Each climb step in TRYALLOC and UNMARK is a while loop for updating the state of a node that completes via a successful `CAS`. Similarly, the climb steps within FREENODE can complete if the successful `CAS` is performed by either the invoking thread A or by a concurrent thread B. Hence, the claim follows. □

Theorem 4.2. *S1 is satisfied.*

Proof. By line A5, NBALLOC selects for the allocation of an s-byte request the deepest level L (within the tree) such that $s \leq \texttt{min\_size} \times 2^{\texttt{base\_level}-L}$. Then, it iteratively calls the TRYALLOC function passing the index of a node at level L. Hence (see line T2), it attempts to set the `BUSY` flag via `CAS` exclusively on the node identified by the input index, namely a node at level L. Therefore, no other node of the tree at levels different from L will ever be set as fully occupied when an allocation request asking for s bytes is run. Hence, no chunk larger than the smallest size useful to serve the s-byte request is allocated and the claim follows. □

Theorem 4.3. *S2 is satisfied.*

Proof. By the structure of NBFREE, FREENODE and UNMARK, we know that a release operation will only act on nodes of the tree (or a subset of them, if an early-stop occurs while climbing, see lines F15-F17 and lines U8-U9) that are included in the path between the node to be released, which we denote as x, and an ancestor standing at `base_level`. In fact, FREENODE and UNMARK perform climbs along this path. The operations done on the nodes belonging to this path are the following ones. The node to be released is marked with 0 in its state, see line F20. The other nodes in the path are initially marked as left/right coalescing, see line F13, hence their status is not set to 0 by this operation. Successively, their status is set to 0 via `CAS`, see line U11, if they are found to be still coalescing. Also, this operation is successful only if no other `CAS` carried out by concurrent operations leads to a conflict. Hence, when the state of a node different from x is set to zero during a release, this can never happen for nodes that have status bits different from `COAL_LEFT` or `COAL_RIGHT` set to 1 (by a concurrent or an already executed `CAS`). Therefore, no partially or fully occupied node different from x is ever released (marked with 0) by the release operation. Hence, the claim follows. □

Corollary 4.4. *Release operations effectively coalesce memory.*

Proof. This can be inferred by the proof of Theorem 4.3. In fact, the left (right) coalescent and occupancy bits of an ancestor node n of the just released node can be set to 0 if the coalescing bit is still set during the UNMARK execution. Consequently, if the opposite right (left) bits were already set to 0, the whole state of p results equal to 0. It means that a subsequent allocation can fully acquire p. Hence, the claim follows. □

Theorem 4.5. *S3 is satisfied.*

Proof. Without loss of generality, let us suppose that the allocation has targeted a generic level L of the buddy system, and a node x. From Lemmas 4.5 and 4.6 the allocated memory chunk is contiguous and aligned with respect to the target size, hence it fully stands in a specific sub-tree of the buddy system, starting at level L. This sub-tree handles chunks that are fully disjoint with respect to those associated with other sub-trees starting at the same level L. Let us suppose by contradiction that the allocated chunk corresponding to node x is already allocated, either partially or fully. In this case, by Lemma 4.7, a concurrent, or already completed, allocation has already set the status of all the nodes that stand on the tree on a sub-path included between node x and the ancestor node at `base_level`. Hence, the `CAS` instructions at line T2 or line T15 must have failed while attempting to allocate node x, since they also attempt to update the state of the traversed nodes, and the allocation by NBALLOC would have failed, which contradicts the assumption. Hence, the claim follows. □

Theorem 4.6. *S4 is satisfied.*

Proof. By hypothesis we know that there is enough memory to satisfy the request. By Theorems 4.3-4.7, it follows that some node at the target level has its state set to 0 and all of its ancestors are either completely free or fragmented. Since the thread executes in isolation by hypothesis, it finds the free node during the scan of the tree (see lines A6-A14 of Algorithm 2). Finally, it (eventually) updates the targeted node and its ancestors according to the TRYALLOC protocol (see Algorithm 3) with an atomic `CAS` that cannot fail because of the absence of concurrency imposed by hypothesis. Hence, the claim follows. □

Finally, let us consider a release operation issued with illegal parameters. Line A10 of Algorithm 2 (NBALLOC routine) writes the index of the allocated node within an entry in the `index` array associated with the lowest-addressed leaf of the allocated chunk of memory. During a release phase, the address A (regardless its alignment) issued as parameter is used to compute the lowest address of the leaf L including A (see line F2 of Algorithm 4). If A is not a base address of a previously allocated memory chuck, the entry of the `index` array corresponding to L should simply result in an invalid index. To this aim, we would just need to reset the entry within the above-mentioned array to 0—recall that such an index does not correspond to any node within the tree—and

check it upon releases. However, such an update should be performed with an atomic CAS paired with a tag to avoid ABA problems.

5 Experimental Results

5.1 Setup

For the assessment we used 4 different test scenarios, two of which have been taken from the literature, while the other two have been appositely devised for this study. The first is the Linux Scalability test [28], where threads execute a fixed-length burst of allocations followed by a burst of releases. The second is called Thread Test [14]. It is based on threads behaving as in Linux Scalability, but the burst length is reduced proportionally with respect to the thread count. The third test (which is one of the two we devised), will be referred to as Cache Test. It makes each thread execute an allocation immediately followed by a release operation. Finally, the fourth test (still devised by us for this study), which we call Constant Occupancy, is based on having each thread initially allocating a pool of chunks of different sizes, with larger amount of allocations bound to smaller chunk sizes, and then performing deallocations/allocations by randomly selecting the element to be deallocated—hence the corresponding deallocation size—and using this same size for the subsequent allocation. Compared to the others, this test more prominently uses allocations/deallocations involving chunks of different sizes and tends to keep constant the factor of occupancy of the buddy system. All the tests have been carried out on a 64-bit NUMA HP ProLiant server with four 1.9GHz AMD Opteron 6168 processors and 128 GB of RAM. Each processor has 12 cores, for a total of 48 CPU-cores. The operating system is Linux, with kernel version 5.4.0.

5.2 Comparison with the Linux-kernel Buddy System

We have compared the performance of our non-blocking buddy system (NBBS) with the Linux buddy system (denoted as `linux-bs`), which is one of the most widely used back-end memory allocators around the world. Also, the Linux buddy system already embeds mechanisms that reduce the impact of synchronization on concurrent allocation/release operations. In particular, it is based on a multi-list data structure, and exploits per-list spin locks for handling concurrency. Furthermore, for single page allocations, the Linux buddy-system API is internally redirected to a cached allocator that delivers already allocated memory (therefore memory already pre-reserved via the actual buddy-system internal mechanisms). These cached allocators are per-CPU lists, which further helps saving conflicts when allocations/releases of single page buffers occur. Overall, using the Linux buddy system as the competitor helps us assessing our proposal comparing it with an extremely well conceived allocator already oriented to concurrency.

Linux handles multiple buddy-system instances associated with the different NUMA nodes in the hardware architecture—and the different memory zones in these nodes, such as NORMAL or DMA/DMA32 zones. In our tests, we compared the performance of an individual NORMAL instance of the Linux buddy-system allocator with one instance of our NBBS. We already hinted that replicating the (core) memory allocator to reduce the impact of concurrency on each single instance is a technique fully orthogonal to the objectives of our proposal and can be still exploited in combination with NBBS. Overall, in this study we focus on single instance tests since they are useful to capture the actual benefits by our proposal, which can be then reflected on a larger scale when replicating the (core) memory allocator.

In our tests, we have assessed both the original and the 4-level optimized versions of NBBS, which we denote as `1lvl-nbbs` and `4lvl-nbbs` respectively, both running with the free-slot optimization discussed in Section 3.6.1.

We developed Linux loadable modules implementing the logic for the tests related to Linux Scalability, Thread Test, Cache Test and Constant Occupancy. NBBS has been embedded in the loadable modules, so that both the compared allocators fully operate at kernel level. This allows fairness in the comparison. Overall, the tests have been based on kernel level threads which interact with the Linux buddy system via the `__get_free_pages` and `free_pages` kernel-level functions, or equivalently with the API of NBBS. The GFP flags used for the allocation API are GFP_KERNEL and GFP_ATOMIC, that define if threads are allowed or not to sleep while serving memory requests. The GFP_ATOMIC configuration is also called `linux-bs-at`. Our target machine has 8 NUMA nodes, so the Linux kernel handles 8 instances of a buddy allocator in parallel for managing NORMAL zones. To test the performance of a single allocator instance—for the reasons we explained before—we set the memory-policy of the threads activated within the Linux module so as to bind the allocations towards the same buddy-system instance, namely, instance 1 which is equipped with 16GB sized NORMAL zone (conversely, the same zone in instance 0 maintains 12GB of memory). The thread count has been varied from 1 to 48. Thanks to this, we can stress the allocators with a higher level of concurrency with respect to the one provided by an individual NUMA node, which is equipped with 6 cores. On the other hand, we are still able to evaluate their behavior with low/no contention (as with conventional workload with multi-instance configuration of the memory allocator). We have configured the compared allocators to manage the same amount of memory, and with the same granularity of minimum and maximum chunk size—the minimal size is set to 4KB and the maximal size is set to 4MB. Finally, we have ensured that each benchmark is configured in a way that no-failed allocations might occur during the experiments to assess that performance metrics are comparable across different allocators.

In Figures 7-10 we report data for all the tested scenarios. Beyond scaling the number of threads to test with different concurrency levels, for each scenario we also used a set of different allocation/release sizes ranging from 4KB bytes to 256KB bytes. For the Constant Occupancy test these values represent the minimum size among those managed, while the maximum size is set to be 8 times larger.

Generally speaking, the data show a clear gain by NBBS, which increases when increasing the thread count. In other words, the Linux-kernel buddy system does not scale regardless of the GFP flags used for the allocation requests (`linux-bs-at` has a slightly worse performance),

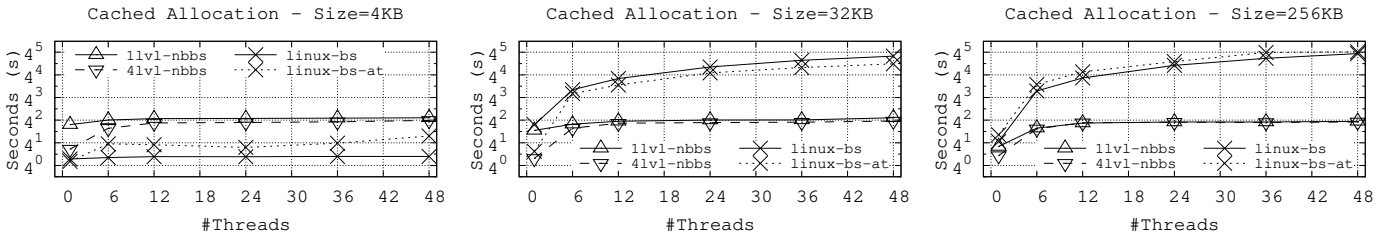


Fig. 7. Execution times - Cache Test benchmark.

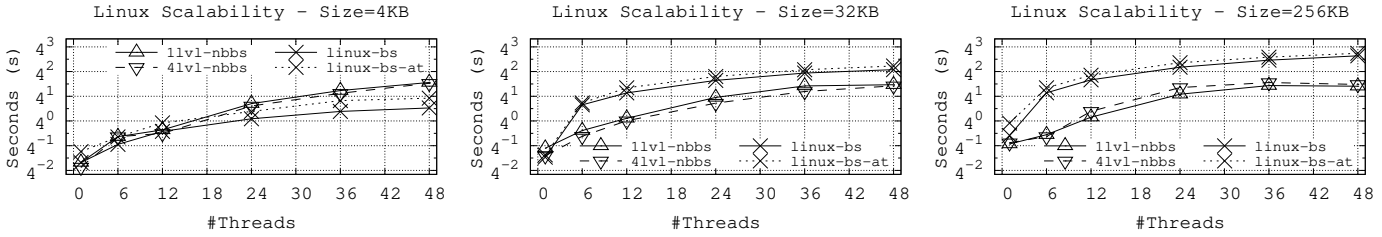


Fig. 8. Execution times - Linux Scalability benchmark.

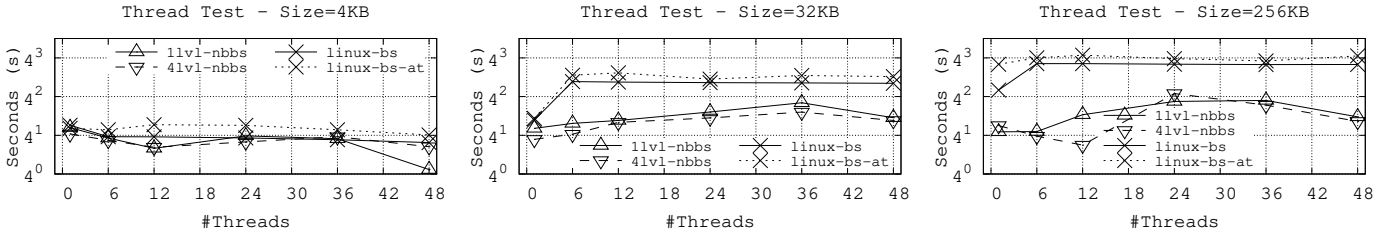


Fig. 9. Execution times - Thread Test benchmark.

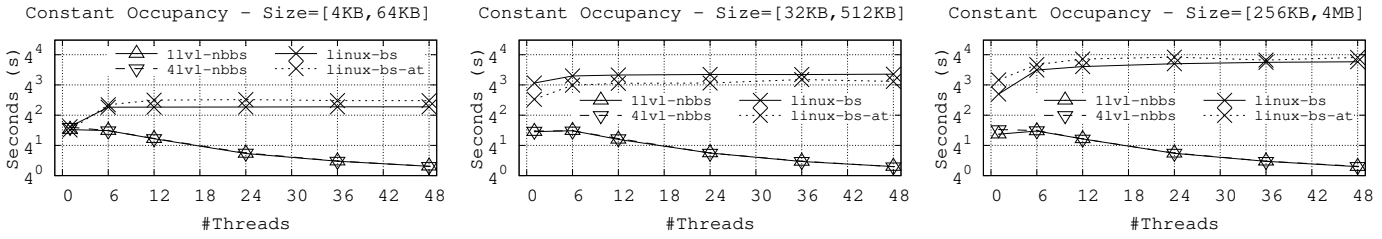


Fig. 10. Execution times - Constant Occupancy benchmark (recall that 8, 128 and 1024 are minimum sizes for the allocations - the maximum corresponding sizes are 128, 2048 and 16384 bytes as specified by the test structure).

while our solution provides an improved scalability and conflict resiliency. Also, both the 1-level and 4-level organizations of our non-blocking buddy system have similar performance. The motivations behind their similar behavior are linked to the tradeoff between the number of atomic RMW instructions and the probability of conflicts. Specifically, in scenarios where a single allocation is followed by a single release, the lower level of fragmentation of the buddy system allows for less conflicting operations when working with RMW instructions at individual levels, as in the 1-level organization. On the other hand, the 4-level optimization still provides advantages in some of the configurations with reduced thread counts, mostly thanks to the reduced amount of steps required to update the buddy system.

The results for Cached Allocation (see Figure 7) show that for sizes larger than 4KB, NBBS provides a speed up with respect to `linux-bs` (or `linux-bs-at`) from 4x to 64x. The 4-level optimization shows its benefits when running with a single thread thanks to its reduced critical-path cost. These advantages are clearly less relevant when contention increases or the operations start from upper levels of the tree (e.g., with allocations of 256KB). Conversely, when the requests target the size of a single 4KB page, `linux-bs` outperforms NBBS. However, this is expected for two main reasons. On the one hand, as noted before, the `linux-bs` adopts per-CPU lists[3] for managing single-page allocations, thus it is capable to satisfy 4KB requests via pre-reserved/per-CPU memory, without any execution of actual buddy-management operations. On the other hand, `linux-bs` also resorts to *lazy coalescing* of nodes, namely the merge of pages is not executed on-the-fly upon memory-release operations. NBBS can be anyhow combined with this kind of optimization, which is actually orthogonal with respect to the target of our proposal. In any case, for all the other allocation sizes the benefits by NBBS show the real power of the non-blocking approach we devised, which is capable of performing coalescing and splitting in fully non-blocking fashion while performing regular alloca-

3. See `https://elixir.bootlin.com/linux/v5.4/source/mm/page_alloc.c#L3265`

TABLE 2
Speedup provided by NBBS with respect to both front-end and back-end allocators.

		1-level NBBS						4-level NBBS					
		libc	sfmalloc	lrmalloc	linux-bs	dsa-wf	cmalloc	libc	sfmalloc	lrmalloc	linux-bs	dsa-wf	cmalloc
4KB	Cached Allocation	0.60	0.29	0.30	0.10	0.62	15.48	0.81	0.40	0.40	0.20	0.84	19.66
	Costant Occupancy	0.90	0.80	33.07	7.36	0.85	10.12	0.88	0.78	32.72	7.26	0.83	9.99
	Linux Scalability	7.52	5.81	1.11	0.59	7.95	4.04	8.63	6.68	1.33	0.69	9.10	4.60
	Thread Test	2.64	3.36	0.80	1.36	2.59	10.04	2.24	3.37	0.93	1.20	2.19	7.79
32KB	Cached Allocation	0.68	0.33	56.72	21.83	0.70	17.01	0.87	0.41	67.59	26.84	0.88	20.46
	Costant Occupancy	0.94	-	146.72	33.49	0.90	7.89	0.92	-	145.88	33.24	0.89	7.84
	Linux Scalability	10.07	6.09	15.78	2.66	10.36	1.94	12.81	7.79	19.80	3.40	13.18	2.46
	Thread Test	2.42	4.52	20.35	3.02	2.37	2.42	3.43	6.04	25.23	3.83	3.35	3.08
256KB	Cached Allocation	0.96	-	149.08	54.63	0.93	8.21	0.92	-	147.20	53.88	0.89	8.06
	Costant Occupancy	0.94	-	146.72	33.49	0.90	7.89	0.92	-	145.88	33.24	0.89	7.84
	Linux Scalability	12.72	-	16.76	5.74	12.84	1.88	11.17	-	14.61	5.10	11.28	1.66
	Thread Test	2.54	-	19.63	6.00	2.65	2.32	2.50	-	27.57	8.38	2.59	3.1

Legend: $(0, 0.5]$ | $(0.5, 0.9]$ | $(0.9, 1.1)$ | $(1.1, 2.0]$ | $(2.0, +\infty)$

tions/releases, still in non-blocking manner.

A similar evidence emerges for the Linux Scalability test (see Figure 8), which has a similar access pattern, but uses bursts of allocations and releases. Also in this case, the `linux-bs` gains from the adoption of per-CPU lists, providing up to 4x performance improvements at maximum thread count compared to NBBS. However, this trend is reversed when using larger sizes for allocations, even with a reduced level of concurrency, again thanks to the higher effectiveness of the NBBS internal mechanisms compared to `linux-bs` in face of concurrent accesses. With Thread Test (see Figure 9), the differences between NBBS and `linux-bs` with page-size allocations disappear, and NBBS still provides advantages (4x speed up) for large allocations and with thread counts greater than or equal to 6.

As a central aspect, whenever the Linux buddy system cannot rely on per-CPU lists—like when operating allocations/releases at different levels—its performance significantly degrades. The Constant Occupancy test (see Figure 10) stresses this weakness by supplying requests spanning multiple (four) levels and showing the maximum benefits achieved by our NBBS, which provides a minimum of 4x and up to 64x speed up.

5.3 Comparison with Other Allocators

We compared our solution with additional front-end and back-end allocators. On the front-end side, we have considered the blocking `libc` allocator (included within the glibc library) and two non-blocking (lock-free) allocators: `sfmalloc`[4] [29] and `lrmalloc`[5] [19]. As additional back-end allocators, we considered a wait-free dynamic-storage allocator [21] (denoted as `dsa-wf`) and the lock-free `cmalloc`[6] [23] (denoted as COA). Since there is no available implementation of `dsa-wf`, we implemented and made it publicly available[7].

Table 2 reports in each cell the speedup, averaged across all thread counts, delivered by our solutions with respect to the other allocators (punctual data are provided in the supplemental material). Light and dark green (red) cells indicate at least 1.1x and 2x speedup (slowdown). A white cell means that the speedup (slowdown) is bounded by 1.1x. Since the `sfmalloc` allocator crashes if multiple requests are concurrently issued targeting memory sizes larger than one page, we denoted such cases with a '-' symbol in Table 2. Here, we can see that front-end allocators are a better choice whenever their front-end optimizations (e.g. per-thread caches) can be exploited. Otherwise, our NBBS represents a preferable choice.

As the data show, most of the cells are dark green. In fact, NBBS definitely outperforms the Linux-kernel buddy system whenever coalescing/splitting of blocks is a highly recurrent activity of the benchmark (i.e., `linux-bs` cannot exploit its per-CPU caches). Such benefits are still evident when comparing with `dsa-wf`. In fact, we provide at least 2x speed up in both Linux Scalability and Thread Test and our average performance loss is limited by 38% (1.61x slowdown) for the remaining benchmarks. Finally, NBBS outperforms `cmalloc`. This is somehow expected because of the general-purpose nature of the adversary, which resorts to additional memory management (allocation and garbage collection) for the dynamic data structures it is based on.

6 Conclusions

We have presented a non-blocking approach for memory allocators based on the well-known buddy-system specification. To date, this is the first practical proposal of a buddy system that jointly supports allocation, release, and split/coalescing operations, all implemented in non-blocking fashion. A proof of correctness of our buddy system is also provided, in terms of liveness and safety of the memory allocation/release operations it supports. Moreover, we have presented optimizations aimed at reducing the number of executed RWM (Read-Modify-Write) atomic instructions by a factor of 4, the memory overhead by a factor of 15 and the impact of free slot search. Finally, we have shown the effectiveness of our solution experimentally, by comparing its performance to one of the most used and optimized buddy-system implementations, namely the Linux-kernel buddy system.

4. Downloaded from `https://github.com/jeffhammond/sfmalloc` on November 1 2020

5. Downloaded from `https://github.com/ricleite/lrmalloc` on November 1 2020

6. Downloaded from `https://github.com/ricleite/coa` on November 1 2020

7. `https://github.com/HPDCS/dsa-wf`

References

[1] GNU.org. GNU C Library.

[2] M. Gorman, *Understanding the Linux Virtual Memory Manager*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2004.

[3] J. Bonwick, "The slab allocator: An object-caching kernel memory allocator," in *Proceedings of the USENIX Summer 1994 Technical Conference on USENIX Summer 1994 Technical Conference - Volume 1*, ser. USTC94. USA: USENIX Association, 1994, p. 6.

[4] M. Herlihy, "Wait-free synchronization," *ACM Trans. Program. Lang. Syst.*, vol. 13, no. 1, pp. 124–149, Jan. 1991.

[5] T. L. Harris, "A pragmatic implementation of non-blocking linked-lists," in *Proceedings of the 15th International Conference on Distributed Computing*, ser. DISC '01. London, UK, UK: Springer-Verlag, 2001, pp. 300–314.

[6] P. Ramalhete and A. Correia, "POSTER:* A wait-free queue with wait-free memory reclamation*," in *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Austin, TX, USA, February 4-8, 2017*, 2017, pp. 453–454.

[7] C. Purcell and T. Harris, "Non-blocking hashtables with open addressing," in *Distributed Computing, 19th International Conference, DISC 2005, Cracow, Poland, September 26-29, 2005, Proceedings*, 2005, pp. 108–121.

[8] M. Ianni, A. Pellegrini, and F. Quaglia, "Anonymous readers counting: A wait-free multi-word atomic register algorithm for scalable data sharing on multi-core machines," *IEEE Transactions on Parallel and Distributed Systems*, vol. 30, no. 2, pp. 286–299, Feb 2019.

[9] A. Larsson, A. Gidenstam, P. H. Ha, M. Papatriantafilou, and P. Tsigas, "Multiword atomic read/write registers on multiprocessor systems," *Journal of Experimental Algorithmics*, vol. 13, no. 1, p. 1.7, 2009.

[10] N. G. Bronson, J. Casper, H. Chafi, and K. Olukotun, "A practical concurrent binary search tree," *SIGPLAN Not.*, vol. 45, no. 5, pp. 257–268, Jan. 2010.

[11] A. Natarajan and N. Mittal, "Fast concurrent lock-free binary search trees," in *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '14. New York, NY, USA: ACM, 2014, pp. 317–328.

[12] J. Lindén and B. Jonsson, *A Skiplist-Based Concurrent Priority Queue with Minimal Memory Contention*. Cham: Springer International Publishing, 2013, pp. 206–220.

[13] R. Marotta, M. Ianni, A. Pellegrini, and F. Quaglia, "A conflict-resilient lock-free calendar queue for scalable share-everything pdes platforms," in *Proceedings of the 2017 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*, ser. SIGSIM-PADS '17. New York, NY, USA: ACM, 2017, pp. 15–26.

[14] E. D. Berger, K. S. McKinley, R. D. Blumofe, and P. R. Wilson, "Hoard: A scalable memory allocator for multithreaded applications," *SIGPLAN Not.*, vol. 35, no. 11, pp. 117–128, Nov. 2000.

[15] D. Dice and A. Garthwaite, "Mostly lock-free malloc," *SIGPLAN Not.*, vol. 38, no. 2 supplement, pp. 163–174, Jun. 2002.

[16] M. M. Michael, "Scalable lock-free dynamic memory allocation," in *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, ser. PLDI '04. New York, NY, USA: ACM, 2004, pp. 35–46.

[17] A. Gidenstam, M. Papatriantafilou, and P. Tsigas, "Nbmalloc: Allocating memory in a lock-free manner," *Algorithmica*, vol. 58, no. 2, pp. 304–338, Oct 2010.

[18] S. Seo, J. Kim, and J. Lee, "Sfmalloc: A lock-free and mostly synchronization-free dynamic memory allocator for manycores," in *2011 International Conference on Parallel Architectures and Compilation Techniques*, Oct 2011, pp. 253–263.

[19] R. Leite and R. Rocha, "Lrmalloc: A modern and competitive lock-free dynamic memory allocator," in *High Performance Computing for Computational Science – VECPAR 2018*, H. Senger, O. Marques, R. Garcia, T. Pinheiro de Brito, R. Iope, S. Stanzani, and V. Gil-Costa, Eds. Cham: Springer International Publishing, 2019, pp. 230–243.

[20] I. Buck, C. I. Rodrigues, S. Jones, X. Huang, and W. mei Hwu, "Xmalloc: A scalable lock-free dynamic memory allocator for many-core machines," *2010 IEEE 10th International Conference on Computer and Information Technology (CIT)*, vol. 00, pp. 1134–1139, 2010.

[21] P. Stellwag, J. Krainz, and W. Schröder-Preikschat, "A wait-free dynamic storage allocator by adopting the helping queue pattern," in *Proceedings of the 9th IASTED International Conference*, vol. 676, no. 052, 2010, p. 79.

[22] G. E. Blelloch and Y. Wei, "Concurrent fixed-size allocation and free in constant time," 2020.

[23] R. Leite and R. Rocha, "A lock-free coalescing-capable mechanism for memory management," in *Proceedings of the 2019 ACM SIGPLAN International Symposium on Memory Management*, ser. ISMM 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 7988.

[24] K. Fraser, "Practical lock-freedom," Ph.D. dissertation, University of Cambridge, 2004.

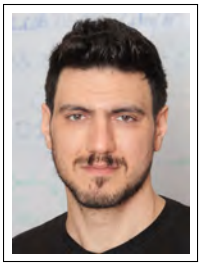
[25] M. M. Michael, "Hazard pointers: Safe memory reclamation for lock-free objects," *IEEE Trans. Parallel Distrib. Syst.*, vol. 15, no. 6, pp. 491–504, Jun. 2004.

[26] D. E. Knuth, *The Art of Computer Programming, Volume 1 (3rd Ed.): Fundamental Algorithms*. USA: Addison Wesley Longman Publishing Co., Inc., 1997.

[27] M. Herlihy and N. Shavit, "On the nature of progress," in *Proceedings of the 15th International Conference on Principles of Distributed Systems*, ser. OPODIS'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 313–328.

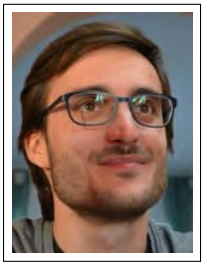
[28] C. Lever and D. Boreham, "Malloc() performance in a multithreaded linux environment," in *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ser. ATEC '00. Berkeley, CA, USA: USENIX Association, 2000, pp. 56–56.

[29] D. Lee, J. Kim, U. Kim, Y. I. Eom, H. K. Jun, and W. T. Kim, "A fast lock-free user memory space allocator for embedded systems," in *ICCSA Workshops*. IEEE Computer Society, 2011, pp. 227–230.



Romolo Marotta received the PhD in Computer Engineering at Sapienza, University of Rome in Italy, in 2020 and is a member of the High Performance and Dependable Computing Systems research group at the same institution. He received the Bachelor's degree in Computer Engineering in 2012 and the Master's degree in 2016. His research activities focus mainly on parallel and concurrent computing, and parallel simulation.

Mauro Ianni has received the PhD in Computer Engineering at Sapienza, University of Rome in 2019, and is a member of the High Performance and Dependable Computing Systems research group at the same institution. He achieved the Bachelors degree in Computer Engineering in 2012 and the Masters degree in Distributed Systems and Computer Architectures in 2015. His research activities focus on concurrent programming and algorithms.



Alessandro Pellegrini received a B.S. degree and a M.S. degree and a PhD in Computer Engineering at Sapienza, University of Rome. His main research area is on parallel and distributed architectures and applications, where he has published more than 50 technical articles. In 2015 he won the Sapienza prize for the best PhD thesis of the year. He has worked as a researcher at some national and international research centers (CINI, CINFAI and IRIANC).

Francesco Quaglia received his MS in Electronic Engineering in 1995 and his PhD in Computer Engineering in 1999, both from Sapienza University of Rome, where he has worked as Assistant and Associate Professor from September 2000 till June 2017. Since then, he works as a Full Professor at the University of Rome Tor Vergata. His research interests include parallel and distributed computing systems and applications, operating systems, high performance computing and cyber-security.