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

Romolo Marotta, Mauro Ianni, Andrea Scarselli, Alessandro Pellegrini
Sapienza, University of Rome
{marotta, mianni}@dis.uniroma1.it andreascarse@gmail.com pellegrini@dis.uniroma1.it

Francesco Quaglia
University of Rome "Tor Vergata"
francesco.quaglia@uniroma2.it

***Abstract*—Common implementations of core memory allocation components, like the Linux buddy system, 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. Conflict detection relies on atomic Read-Modify-Write (RMW) machine instructions. Beyond improving scalability and performance, our solution can also avoid wasting clock cycles for spin-lock operations by threads that could in principle carry out their memory allocations/releases in full concurrency.**

I. Introduction

In standard libraries or in an Operating System (OS), memory allocation is de-facto a *shared-data management* problem. In fact, allocators deal with the issue of managing memory buffers in face of requests that can be concurrently issued by multiple threads. This requires thread-coordination mechanisms in order to guarantee the coherence of the state of the memory allocator at any time.

A classical coordination approach, which is widely used in allocators, consists in using spin-locks. With this approach, the manipulation of shared data representing the state of the memory allocator is implemented as a critical section, hampering scalability. This is a relevant issue, since the level of concurrency is increasingly exacerbated because of the modern-hardware trend towards multi/many-core technologies.

The historical approach for reducing the impact of thread coordination (and the associated costs) on performance with concurrent memory allocations/releases is based on either: a) pre-reserving *arenas*, namely memory segments, for each individual thread—this is what typically happens in user-space allocators [1], [2]— or b) the usage of intermediate allocation services, called *cached allocators*—as for the case of OS-kernel allocation services based on quick-lists [3].

Both approaches aim to reduce the likelihood of inducing large volumes of concurrent accesses to the core allocator that is in charge of ultimately delivering memory, either logical or physical. As for 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. As for point (b), cached allocation diminishes the pressure of concurrent accesses to the core allocator—e.g. a kernel-level buddy system—by having upper-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 allocations or even SLAB allocations. In such a case, concurrent threads performing memory allocations require coordination 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 tackled 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.

Generally speaking, literature approaches make memory management be extremely layered: the upper the layer, the more it satisfies specific requests. In this context, we can define *back-end* and *front-end* allocators. The former, previously denoted as core allocators, handle the lowest level of memory management. The latter are built on top of the back-end allocator with the goal of reducing the pressure of (concurrent) accesses to it and 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 pressure to core allocators by designing front-end allocators, e.g., adopting (a), (b), or multiple-instance approaches. In particular, our contribution is the design of a non-blocking back-end allocator instance implementing the buddy-system specification, where concurrent allocations/releases are not coordinated via spin-locks.

In our approach, coordination and the guarantee of an always-coherent state of the buddy system are achieved by only relying on individual 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 of them to be retried as in the classical non-blocking algorithmic paradigm devised in [4]. However, if conflicts do not materialize, then our proposal saves the latency that would be spent by lock-based approaches, which temporarily block concurrent operations a-priori of their execution.

Clearly, 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 simply optimizes the management of those 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—or by different cached allocators.

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

The remainder of this article is structured as follows. In Section II we discuss related work. The non-blocking buddy system is presented in Section III. Experimental data are provided in Section IV.

II. Related Work

The seminal literature work providing the base-ground for non-blocking coordination in the management of shared data is [4]. This work has opened the path towards the design and implementation of algorithms that can well fit the scalability requirements imposed by modern (large-scale) multi-core machines. Also, the avoidance of lock-based coordination 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 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].

While many solutions have been devised in order to reduce the negative impact of concurrency and synchronization in memory allocation/deallocation by relying on pre-reserving or caching, no one fully faces the problem of concurrent accesses to back-end allocators. The work in [14] provides non-blocking capabilities of memory allocations with high probability, just depending on the pattern of memory allocations/releases and the overall memory usage. 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 presented in [15], [16], [17]. Like the previous work, these proposals do not address the problem of avoiding blocking allocations/releases in scenarios where a same allocator instance can be concurrently accessed by multiple threads. This issue is partially alleviated in [19], where a non-blocking stack data structure is used to post memory across different threads within the pre-reserving scheme.

The solution in [18] 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. 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.

We note that 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 into physical memory of logical pages 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 to by our proposal can be found in [20]. Here the authors present a concurrent non-blocking memory allocator relying on the so called *helping* strategy—where threads help each other

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

trying to avoid blocking scenarios. This solution is based 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, this solution is not able to early detect memory fragmentation, thus possibly leading to a large amount of false positives and unnecessary retries.

III. The Non-blocking Buddy System

Before introducing our solution, a preliminary notion of buddy-system specification is given. A buddy system is a memory allocator that divides a contiguous memory region into partitions, namely memory blocks, by splitting recursively it into halves. These partitions are always contiguous, thus minimizing external fragmentation. In order to satisfy a memory allocation request, it searches for a memory block large enough to satisfy the request.

Every memory block has an order, which is an integer ranging from 0 to a specified upper limit. The size of a block of order n is proportional to 2^n, so that blocks are exactly twice the size of blocks that are at one order lower.

The peculiarity of a buddy system is that memory allocation requests can be satisfied by: i) returning a block if already available; ii) splitting higher-order free blocks into smaller ones; iii) merging lower-order free blocks into a larger one. The binary arrangement of blocks allows to simplify the split/merge of blocks making them very fast thanks to the exploitation of bit-wise operations. For example, given the address of a block of a given order, finding the address of the relative buddy is a simple arithmetic operation and does not need de-referencing pointers.

In the following four sections, we will discuss the metadata used in our solution, the non-blocking allocation and release procedures, and finally a relevant optimization of our proposal.

A. Basics

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 depth d and its structure is assumed to be already materialized in memory. The root of the tree corresponds to (and keeps track of the state of) the entire memory segment within which allocations will take place. Each child of a node represents a portion (a half) of the parent's chunk of memory, while the leaves represent the state of the minimum allocatable memory chunks, called *allocation units*. 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 form a larger block of memory that exactly corresponds to the parent. This means that, considering an overall size of memory managed by the buddy system equal to `total_memory`, 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 $\texttt{total\_memory}/2^d$.

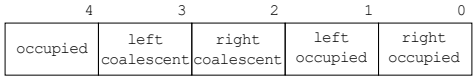

Fig. 1. Node's status bits.

In our non-blocking buddy system, each node in the tree embeds a bit-mask with 5 relevant bits, organized as in Fig. 1. These 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: `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; `left_occupied` and `right_occupied` signal if the branches (left and right, respectively) covered by the node are totally or partially occupied. This means that some allocation request has been served by a node in these sub-trees; `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 the relative sub-trees.

In order to correctly manipulate the status bits while handling concurrent operations, our solution relies on *Read-Modify-Write* (RMW) instructions offered by conventional architectures, like x86. In particular, our operations are based on the *Compare-and-Swap* (CAS) RMW instruction. This instruction updates a given memory location only if its current value is equal to an input value provided to the instruction; it then returns the success (or the failure) of the operation (in this case the instruction is denoted as BCAS) or the value of the memory location before its execution (VCAS).

As in common buddy-system organizations, each allocation of a given size s is mapped to its immediately higher size corresponding to $\texttt{total\_memory}/2^i$ with $i \in [0, d]$. Also, when a node is allocated, its sub-tree 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 the lower-level nodes, which helps us saving atomic RMW instructions for updating the corresponding status bits.

We represent the tree of nodes 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

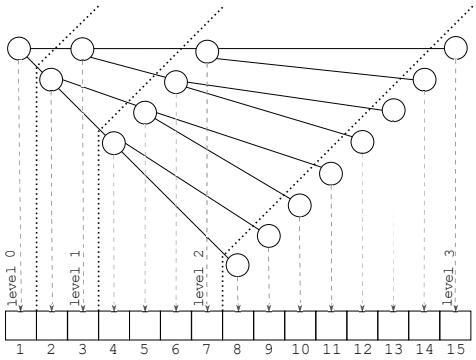

Fig. 2. Array representation of the tree structure.

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 Fig. 2. 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 $\texttt{starting}_\texttt{n}$ and the size $\texttt{size}_\texttt{n}$ of the corresponding memory chunk, as well as the level associated with the node, according to the following rules:

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

$$\texttt{size}_\texttt{n} = \frac{\texttt{total\_memory}}{2^{\texttt{level}_\texttt{n}}} \tag{2}$$

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

We couple the `tree[]` array with another array, called `index[]`. This array is used to keep track of the indexes of the nodes that have been used for serving memory requests, and 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 present our non-blocking allocation/release algorithms by relying on a few additional notations. The maximum 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, and as `max_size` the one maintaining the maximum amount of memory allocatable with a single request (available at level `max_level`)—clearly, $\texttt{max\_size} \leq \texttt{total\_memory}$.

To extract and manipulate the status bits within the nodes of the tree, the following bit-masks are used:

$\texttt{OCC\_RIGHT} = \texttt{0x1}$
$\texttt{OCC\_LEFT} = \texttt{0x2}$
$\texttt{COAL\_RIGHT} = \texttt{0x4}$
$\texttt{COAL\_LEFT} = \texttt{0x8}$
$\texttt{OCC} = \texttt{0x10}$
$\texttt{BUSY} = (\texttt{OCC} \mid \texttt{OCC\_LEFT} \mid \texttt{OCC\_RIGHT})$

Each memory allocation/release operation consists in traversing the tree up to `max_level` in order to correctly manipulate the status bits of the traversed nodes. In fact, these bits need to be (re)aligned to the new state of the buddy system, depending on the type of operation that is 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—this 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, the following status-bit manipulation functions are offered to simplify the exposition:

$clean\_coal(val, child) : val \ \& \ \neg(\texttt{COAL\_LEFT} >> \text{mod}_2(child))$
$mark(val, child) : val \mid (\texttt{OCC\_LEFT} >> \text{mod}_2(child))$
$unmark(val, child) :$
$\qquad val \ \& \ \neg((\texttt{OCC\_LEFT}|\texttt{COAL\_LEFT}) >> \text{mod}_2(child))$
$is\_coal(val, child) : val \ \& \ (\texttt{COAL\_LEFT} >> \text{mod}_2(child))$
$is\_occ\_buddy(val, child) : val \ \& \ (\texttt{OCC\_RIGHT} << \text{mod}_2(child))$
$is\_coal\_buddy(val, child) : val \ \& \ (\texttt{COAL\_RIGHT} << \text{mod}_2(child))$

$clean\_coal(val, child)$ sets to zero the coalescing bit relative to the branch of the child. $mark(val, child)$ sets to one the occupancy bit of the branch of the child. $unmark(val, child)$ sets to zero 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 to one. $is\_occ\_buddy(val, child)$ returns true if the occupancy bit relative to the buddy associated with the child is set to one. $is\_coal\_buddy(val, child)$ returns true if the coalescing bit relative to the buddy associated with the child is set to one. In order to capture whether the child is the right or the left one, all these functions use a two-modulus operation applied to the index of the child.

We additionally use the following status-bit management function to detect whether a node, whose status bits are embedded within the val bit-mask, is currently free:

$is\_free(val) : \neg(val \& \texttt{BUSY})$

This condition is verified when the node itself has not been reserved for some allocation operation (it is not occupied), and none of its sub-trees (left and/or right) has nodes currently reserved for allocations—none of the sub-trees 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 possible memory chunks (and of the corresponding addresses) managed by the buddy system at any level has been delivered for usage.

B. Memory Allocation Algorithm

The non-blocking memory allocation operation is divided in two algorithms, NBALLOC() and TRYALLOC(). The pseudo-code of NBALLOC(), which represents the memory allocation

Algorithm 1 Allocation - Part A

```
A1: procedure NBALLOC(size_t size) return void *
A2:     if size > max_size then
A3:         return false
A4:     end if
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:            end if
A16:        end if
A17:    end for
A18:    return NULL
A19: end procedure
```

Algorithm 2 Allocation - Part B

```
T1: procedure TRYALLOC(index n) return index
T2:     if ¬ BCAS(&tree[n], 0, BUSY) then
T3:         return n
T4:     end if
T5:     current ← n
T6:     while level_current > max_level do
T7:         child ← current
T8:         current ← current>>1
T9:         do
T10:            curr_val ← tree[current]
T11:            if curr_val & OCC then
T12:                FREENODE(n, level_child)
T13:                return current
T14:            end if
T15:            new_val ← clean_coal(curr_val, child)
T16:            new_val ← mark(new_val, child)
T17:        while ¬ BCAS(&tree[current], curr_val, new_val)
T18:    end while
T19:    return 0
T20: end procedure
```

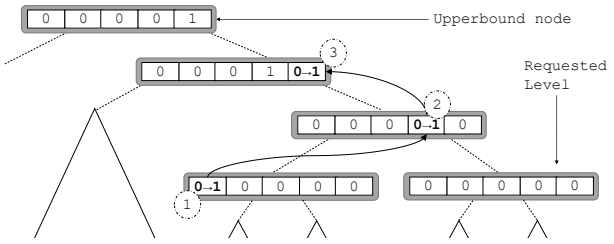

Fig. 3. Visual representation of TRYALLOC operations.

API actually exposed to the user, is reported in Algorithm 1. NBALLOC() takes as input the size of the memory allocation request and, if such amount of memory is available as a set of buddies, it returns the address of a memory chunk big enough to fit the request. If the size exceeds the overall memory allocatable by a single invocation, the allocation fails. Differently, if it is smaller than the minimum amount managed within the buddy system, it is rounded to the allocation unit, namely the size associated with the leaves. In any case, if the request size is legitimate, the target level of nodes to be considered for allocation is obtained by Rule 1 (line A5). Once identified the right level, thus the range of indexes of nodes suitable for the allocation, these nodes are scanned in order to search for a free one. Note that not necessarily such a search has to start from the first node at that level. Rather, 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 "reserve" it by invoking 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, then the allocation operation fails, indicating that the current usage state of the buddy system is not compliant with the issued request.

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 (lines A10). Then, such memory address is returned, indicating a successful allocation.

The pseudo-code of TRYALLOC() is reported in Algorithm 2 and a visual representation of its steps is shown in Fig. 3. It takes as input the index of a node (previously observed as free) and tries to (i) occupy this node, and (ii) propagate the information about the occupancy up to the ancestor node belonging to `max_level`. The former task is carried out by relying on the CAS machine instruction (see line T2 corresponding to step 1 in Fig. 3), which tries to update all the occupancy bits in the status bit-mask of that node, by atomically checking 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 buddy system, 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 instruction succeeds, the procedure continues by traversing the nodes along the path from the one that has been currently occupied towards `max_level`. This traversal is required to update the occupancy bits of the ancestor nodes so as to reflect that the sub-tree has become partially occupied. In this way the corresponding memory chunk will figure out as fragmented into lower level buddies, one of which is occupied, preventing other allocations from occupying an ancestor (a higher-level node). For each node along the path towards `max_level`, the procedure tries to mark its state as left or right occupied, depending on what branch we are backward traversing (e.g. step 2 and 3 in Fig. 3).

This operation is still performed via CAS machine instructions (see line T17) with the peculiarity that, if it fails, it can be retried since the update we are trying to perform can be still coherent with respect to the memory allocation operation we are carrying out and other concurrent operations. As an example, 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.

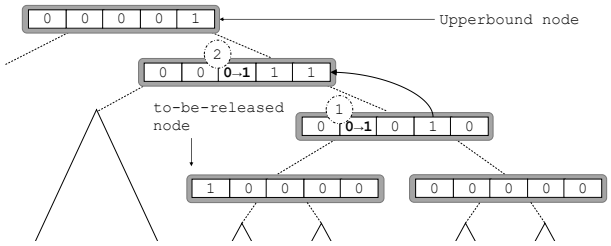

Fig. 4. First phase of the FREENODE (lines F6-F21).

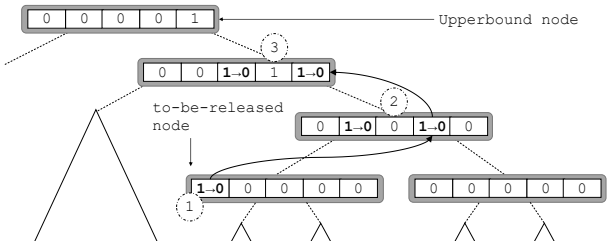

Fig. 5. Second (line F19) and third (UNMARK) phase of the FREENODE.

The only scenario where the failure of the CAS at line T17 indicates that the currently carried out memory operation needs to be aborted is when another operation updated the (fully) `occupied` bit. In this case, another operation has exactly reserved that node—and the corresponding memory chunk—for a concurrent (or already-finalized) memory allocation. Hence, we cannot fragment that chunk to reserve some chunk at lower levels. We also note that, while attempting to left/right occupy a node along the traversal, the corresponding left/right coalescing bit is set to the value 0. As we will clarify while explaining memory release operations, this is required in order to make conflicting releases aware that the branch is involved in a new allocation and cannot be marked as free.

If the max-level node is reached and updated 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 set is found along the path, the allocation fails on the current memory block and nodes updated along the backward traversal have to be reverted. This is done by invoking the FREENODE() procedure in Algorithm 3, 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.

C. Memory Release Algorithm

A memory release operation is composed by three phases. In the first phase, the ancestors of the node to be released are marked as coalescing, in order to notify that the release operation is in place along the corresponding path of the tree. 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. As hinted before, this last update may conflict with a concurrent allocation operation and fail thanks to the CAS semantic—left/right occupancy bits remain set since the memory has been already reused. The first two steps are implemented by the FREENODE() procedure (see Algorithm 3), while the last one is carried out by the UNMARK() procedure (see Algorithm 4).

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

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 `max_level`, making it traverse the tree up to the level corresponding to the maximum allocatable size. In this case, we are releasing a previously allocated node and, thus, the status bits need to be reflected up to the maximum useful level of the tree. Differently, if the procedure is invoked by a failed TRYALLOC() (see line T12), 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 bits of nodes that where involved in the traversal towards the maximum level, which was interrupted because of the discovery of an already occupied node.

While traversing all the nodes up to the upper-bound, FREENODE() atomically sets the coalescing bit of the correct position, via the CAS instruction (see line F13 in Algorithm 3). A visualization of this phase is shown in Fig. 4.

If along this path a buddy is detected as occupied by other allocations, the climb is early arrested, since the corresponding sub-tree cannot be considered free—this is the case of Fig. 4, where the right subtree of the node below the upper-bound is already fragmented. Once the first phase is concluded, FREENODE() can start signaling that the interested node has been released. These steps are sketched in Fig. 5. In particular, the node to be released can be updated by resetting its occupancy bits. This takes place by simply writing zero on its status bits (see line F22 in Algorithm 3 corresponding to step 1 of Fig. 5).

The last phase is responsible of propagating the release of the interested node 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 coalescing and the occupancy bits of the traversed nodes (see steps 2 and 3 of Fig. 5). For each node met, the first step is to verify whether the coalescing bit is still set: 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

Algorithm 3 Memory Release

```
F1: procedure NBFREE(void *addr)
F2:     n ← index[(addr − base_address) / min_size]
F3:     FREENODE(n,max_level)
F4: end procedure
F5: procedure FREENODE(index n, index upper_bound)
F6:     current ← n>>1
F7:     runner ← n
F8:     while level_runner > upper_bound do
F9:         or_val ← COAL_LEFT>>(mod_2(runner))
F10:        do
F11:            cur_val ← tree[current]
F12:            new_val ← cur_val | or_val
F13:            old_val ← VCAS(&tree[current],cur_val,new_val)
F14:        while old_val ≠ cur_val
F15:        if is_occ_buddy(old_val, runner) ∧
F16:                                ¬is_coal_buddy(old_val, runner) then
F17:            break
F18:        end if
F19:        runner ← current
F20:        current ← current>>1
F21:    end while
F22:    tree[n] ← 0
F23:    if level_n ≠ upper_bound then
F24:        UNMARK(n,upper_bound)
F25:    end if
F26: end procedure
```

Algorithm 4 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:            end if
U11:            new_val ← unmark(curr_val, child)
U12:        while ¬ BCAS(&tree[current],curr_val, new_val)
U13:    while (level_current > upper_bound) ∧
U14:                                ¬is_occ_buddy(new_val, child)
U15: end procedure
```

node has already occupied/released that coalescing sub-tree.

Conversely, if the coalescing bit is still set, the procedure tries to clean both the coalescing and the occupancy bits atomically. This operation is done via a CAS in a retry-cycle in order to manage the case in which the coalescing bit has been reset by some concurrent operation, meaning that the resource has already been reused/released. If during this procedure some nodes in the same sub-tree have been allocated, the relative occupancy bits have not to be reset. 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 node releasing and merge buddies up to the higher level, since the chunks associated with higher level nodes 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. This operation, that usually has to be performed explicitly, here is carried out implicitly while performing allocations/releases.

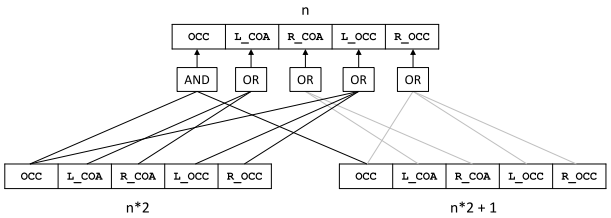

Fig. 6. Derivation of node state by children' state.

D. 4-Levels Optimization

In order to execute an atomic RMW instruction, the CPU core takes an exclusive access to the relative memory cache line, delaying its access to other cores. Reducing the number of RMW instructions executed along the critical path of any thread in a non-blocking algorithm is therefore an important aspect in terms of further performance improvements.

In the previously presented solution, a thread that performs a memory allocation or a memory release has to execute a number of RMW instructions equal to at least the depth of the node involved in the allocation/release with respect to the `max_level` within the tree. However, since 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—it is possible to reduce such amount of RMW instructions by packing a bunch, namely a node and some levels descendant from it, in the same status bit-mask. With this organization a thread is able to update at one time the state of multiple levels with only one RMW atomic instruction. On the other hand, 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 the meanwhile.

Another important consideration is that, given a generic node in the tree, its state can be derived by looking at the state of its children, as shown in Fig. 6. 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 bits of the children nodes. In the same way, the coalescing bits can be derived by 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, since they cannot be seen as individual fragments of memory some of which considerable like 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 the partial occupancy, the fully occupancy of a node can be computed with a logical AND operation on the `occupied` bits of its children.

Such 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 its lower level. Then, considering a word-size equal to w bits, and a status bit-mask of size s bits, we can pack in a single variable a bunch of depth equal to d, with $2^d * s < w$, representing $2^{d+1} - 1$

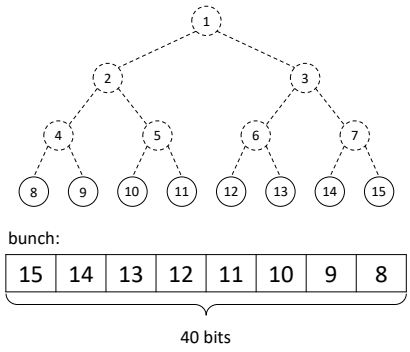

Fig. 7. Array representation of a bunch.

nodes of the original tree. In our case, we are able to manage 4 levels, namely 15 nodes with 8 nodes in the lower level (40 bits), in a single 64-bit word, reducing the number of atomic RMW operations by a factor of 4. A representation of this transformation is shown in Fig. 7.

Of course, this means that, when an operation is performed on a node that is not in a lower level 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 with the same rule used 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 lower level of the bunch. In particular, given a node n with depth d_n and its position inside the bunch b_n, the corresponding bunch-leaf nodes have position p in the bunch equal to:

$$p \in [b_n \cdot 2^{3-mod_4(d_n)}, (b_n + 1) \cdot 2^{3-mod_4(d_n)} - 1]$$

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.

IV. Experimental Results

In our experimental evaluation, the compared allocators do not use pre-reserving, hence they are tested as actual back-end allocation services. This is compliant with our objective, since we focus on improving the efficiency of a back-end memory manager that could be used in combination with any pre-reserving (and thread binding to pre-reserved memory) policy taken from the literature. As already hinted, pre-reserving is an orthogonal technique to improve memory allocation performance with respect to making a back-end allocator more scalable, via non-blocking algorithms as in our approach.

We have compared the performance of our non-blocking buddy system with various alternatives: 1) the buddy allocator in [21] denoted as `buddy-sl`, which is still based on a tree data structure, but synchronizes concurrent accesses using spin-locks, 2) the Linux buddy system, which is instead based on a multi-list data structure, and exploits spin-locks for handling concurrency. Each back-end allocator has been set up in a single-instance configuration, since we are interested in evaluating the benefits/shortcomings of our approach when concurrent requests can actually insist on the same allocation data structure instance, rather than exploiting data-separation via multi-instances. In our tests we have assessed both the original and the 4-level optimized versions of our buddy system, which we denote as `1lvl-nb` and `4lvl-nb`, respectively. Also, we include data related to our own data structure with the variant that, rather than using RMW instructions to make it non-blocking, we synchronize the accesses in a blocking manner by using a unique (global) spin-lock. These configurations, named as `1lvl-sl` and `4lvl-sl` for the two different levels' organizations, respectively, have been included just to study the effects of non-blocking operations against a wider set of blocking solutions, each based on a different implementation.

For the assessment we used 4 different test scenarios, 3 of which have been taken from the literature, and one has been specially devised for this study. The first is the Linux scalability test [22], where threads continuously execute an allocation/release pattern, with fixed size, for at least 20 000 000/`num_threads` times. The second is the Thread Test presented in [17], where threads carry out 10 000/`num_threads` allocations of a given size, and then release the acquired memory, executing this pattern in a cycle of at least 200 steps. The third is the one by Larson in [23], where the behavior of a Web server is simulated, by having a thread reserving memory for (emulated) operations, and then releasing it for usage by other operations. According to its specification, this test is aimed at assessing the operations throughput over a time window of 10 seconds. The fourth—the one we propose—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 it performs 20 000 000/`num_threads` 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 size, 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 equipped with four 2GHz AMD Opteron 6128 processors and 64 GB of RAM. Each processor has 8 cores, for a total of 32 CPU-cores. The used operating system is Linux, with kernel version 3.2.

In Fig. 8-11 we report data for a comparison among all the back-end allocators implemented in user-space. Here, we configured all the tested allocators to manage chunks of minimal size set to 8 bytes, and maximal size set to 16KB. Also, we used different allocation/deallocation sizes ranging from 8 bytes to 1024 bytes. For the Constant Occupancy test these sizes indicate the minimum ones among those managed, while the maximum ones are set to be 16 times larger.

The data show a clear gain of the non-blocking approach when compared to all the other spin-lock-based solutions

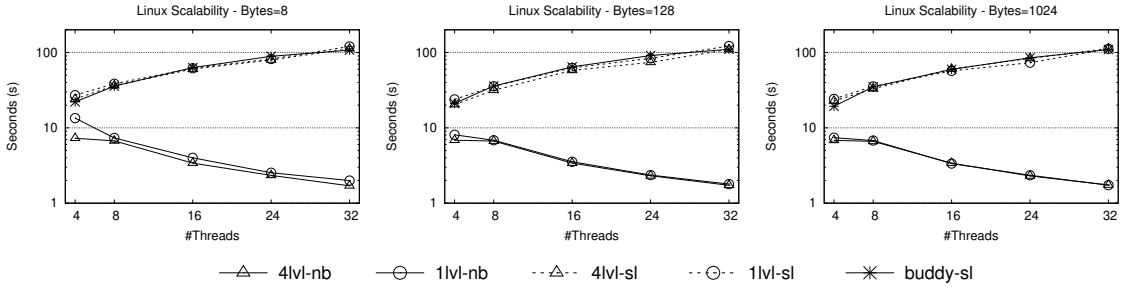

Fig. 8. Execution times - Linux Scalability benchmark.

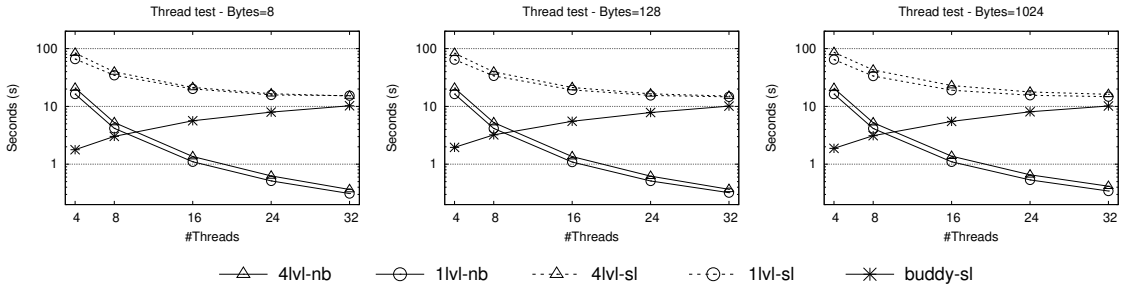

Fig. 9. Execution times - Thread Test benchmark.

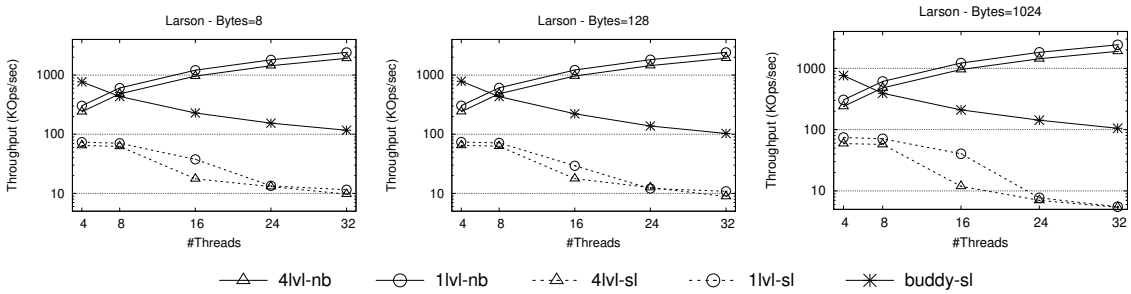

Fig. 10. Throughput - Larson benchmark.

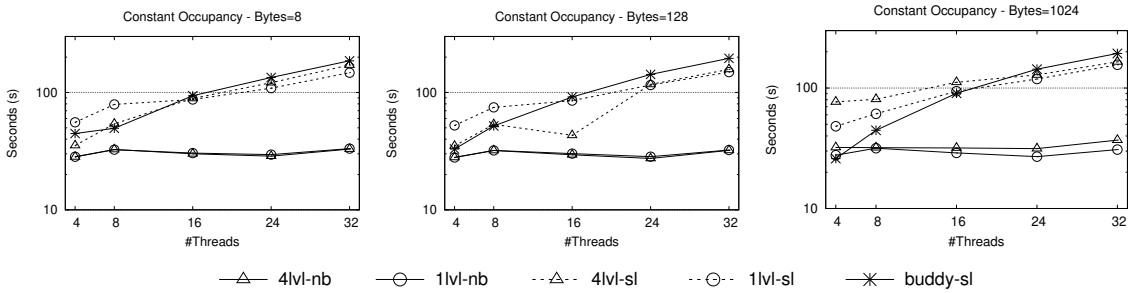

Fig. 11. Execution times - Constant Occupancy benchmark.

especially when increasing the thread count. Also, both the 1-level and 4-level organizations of our non-blocking buddy system have good performance. The motivations behind their similar behaviour are linked to the trade off between the number of atomic operations and the probability of conflicts. Rather, 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. In any case, when running with 32 threads, the non-blocking buddy system provides performance gain ranging from 9% to 95% across all the configurations.

In a second set of experiments, whose results are reported in Fig. 12, we compare the performance of the different allocators this time including the buddy system from version 3.2 of the Linux kernel—to the best of our knowledge, the buddy system is essentially the same in later versions of the Linux kernel. To test with such a kernel level allocator, we developed Linux external modules implementing the logic for the tests related to Linux Scalability, Thread Test and Constant Occupancy. This has been done by relying on kernel level threads which interact with the Linux buddy system via `__get_free_pages` and

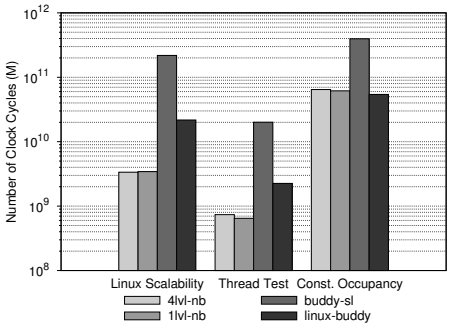

Fig. 12. Comparison with the Linux buddy system.

`free_pages` kernel services. Given that our target machine has 8 NUMA nodes, and hence the kernel handles 8 instances of a buddy allocator in parallel, to test the access performance to the same allocator instance we set the memory-policy of the threads we activated within the Linux module so as to bind the allocations towards the same buddy-system instance (namely, instance 0). We have configured the user-space allocators to manage the same amount of memory as the kernel level buddy allocator, and with the same granularity of minimum and maximum chunk size. The data in Fig. 12 refer to the tests executed when targeting allocations/releases of 128KB chunks (in Constant Occupancy this time the value corresponds to the maximum-size allocatable chunk) with 32 threads. Here, we see that our non-blocking version has performance which is comparable with the Linux buddy system in one case—the Constant Occupancy test—and is definitely better with the other two test settings. In particular, the non-blocking allocator we propose has performance gain that ranges from 71% (as in the 1-level case) to 67% (as in the 4-level case) for Thread Test, and is of the order of 84% for Linux Scalability.

V. Conclusions

We have presented a non-blocking approach for back-end memory allocators based on the well-known buddy-system specification. To date, this is the first practical implementation of a buddy system that jointly supports allocation, deallocation, and coalescing operations, all implemented in non-blocking fashion. Moreover, we have presented an optimization aimed at reducing the number of executed atomic instructions. The experimental assessment has shown the effectiveness of our solution vs both user-space and kernel-space already-optimized allocators. As future work, we plan to integrate our solution with front-end allocators allowing them to interact more frequently with the back-end allocator, thanks to its increased scalability, and to reduce the memory consumption peak.

References

[1] "GNU C Library."

[2] A. Pellegrini, R. Vitali, and F. Quaglia, "Di-DyMeLoR: Logging only dirty chunks for efficient management of dynamic memory based optimistic simulation objects," in *Proceedings - Workshop on Principles of Advanced and Distributed Simulation, PADS*. IEEE, 2009, pp. 45–53.

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

[4] M. P. Herlihy, "Wait-free synchronization," *ACM Transactions on Programming Languages and Systems*, vol. 13, no. 1, pp. 124–149, 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, J. Welch, Ed. Springer Berlin/Heidelberg, 2001, vol. 2180, 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, 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 lock-free O(1) event pool and its application to share-everything PDES platforms," in *20th IEEE/ACM International Symposium on Distributed Simulation and Real Time Applications, DS-RT 2016, London, United Kingdom, September 21-23, 2016*, 2016, pp. 53–60.

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

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] S. Seo, J. Kim and J. Lee, "SFMalloc: A Lock-Free and Mostly Synchronization-Free Dynamic Memory Allocator for Manycores," *Proc. of the 2011 International Conference on Parallel Architectures and Compilation Techniques, PACT 2011*, pp. 253–263, 2011.

[20] P. Stellwag, J. Krainz, and W. Schröder-Preikschat, "A Wait-Free Dynamic Storage Allocator by Adopting the Helping Queue Pattern," in *Parallel and Distributed Computing and Networks (PDCN 2010)*, I. A. of Science and T. for Development (IASTED), Eds., vol. 676, Calgary, AB, Canada, 2010, pp. 79–87.

[21] "https://github.com/cloudwu/buddy/blob/master/buddy.c."

[22] C. Lever and D. Boreham, "Malloc() Performance in a Multithreaded Linux Environment," 2000.

[23] P.-A. Larson and M. Krishnan, "Memory allocation for long-running server applications," in *Proceedings of the 1st International Symposium on Memory Management*, ser. ISMM '98. New York, NY, USA: ACM, 1998, pp. 176–185.

[24] 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.