A Non-blocking Buddy System for Scalable Memory Allocation on Multi-core Machines

Poster extended abstract

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

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

I. Introduction

Memory allocation is a *shared-data management* problem, which requires thread-coordination to guarantee the coherence of the state of the memory allocator. Typical approaches for reducing the impact of thread coordination (and the associated costs) on performance are based on: (a) pre-reserving *arenas* (memory segments) for each individual thread—this is what typically happens in user-space allocators [1], [2], [3]; (b) the usage of intermediate allocation services, called *cached allocators*—as for the case of OS-kernel allocation based on quick-lists [4]; (c) creating data separation on the core allocator via multi-instance approaches, and redirecting the requests towards different instances. These 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 to, e.g., some upper layer allocator.

In this short paper we tackle the issue of scalability of core memory allocators, which is an orthogonal optimization with respect to reducing the pressure to core allocators by (a), (b), or (c). In particular, our contribution is the design of a non-blocking (lock-free) allocator implementing the buddy-system specification, where concurrent allocations/dellocations are not coordinated via spin-locks, but by only relying on individual Read-Modify-Write (RMW) instructions executed along the critical path of allocation/deallocation operations. These instructions are exploited to detect whether concurrent requests have conflicted on the same portion of the allocator metadata.

Our non-blocking solution can be used in combination with any already existing scheme aimed at diminishing the pressure of concurrent accesses to the core allocator, e.g. by introducing multiple instances or combining it with upper level allocators. This is because our unique goal is to provide a memory allocation system that simply optimizes the management of those concurrent accesses. On the other hand, having a more efficient core 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 effectiveness of our proposal. The reader can refer to [5] for details on the algorithmic solutions we used—including their correctness proof—and additional performance data.

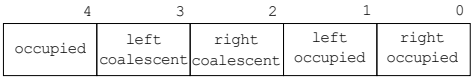

Fig. 1. Node's status bits.

II. The Non-blocking Buddy System

Our non-blocking buddy system keeps track of the state of the memory segment used for serving allocations by the means of an implicit (i.e. packed into an array) static complete binary tree, which 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. 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.

In our non-blocking buddy system, each node embeds a bit-mask with 5 relevant bits, organized as in Figure 1, representing 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`, this flag indicates whether an allocation request has been served by the memory chunk corresponding to that node; (ii) `left_occupied` and `right_occupied`, these flags indicate if some allocation request has been served by a node in the sub-trees (left and right, respectively) covered by the node; (iii) `left_coalescent` and `right_coalescent`, these two flags indicate whether the node is currently in a transient state because of memory-release modifications running on the relative sub-trees.

In order to correctly manipulate the status bits while handling concurrent operations, our solution relies on the atomic *Compare-and-Swap* (CAS) instruction offered by conventional architectures, like x86.

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

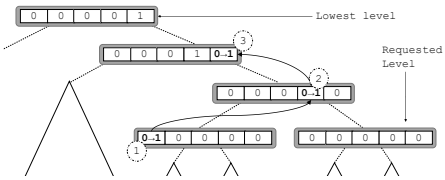

Fig. 2. Memory allocations.

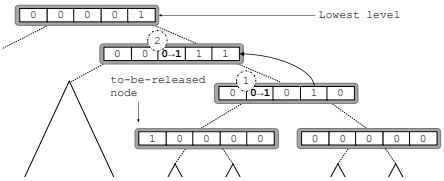

Fig. 3. First phase of a deallocation.

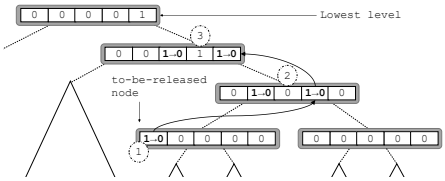

Fig. 4. Subsequent phases of a deallocation.

TABLE I
CLOCK CYCLES COMPARISON WITH THE LINUX BUDDY SYSTEM.

	Thread Test	Linux Scalability
Linux buddy system	225×10^7	2173×10^7
Non-blocking buddy system	64×10^7	343×10^7

a) Memory Allocation Algorithm: The allocation starts by identifying the maximum level that allows to satisfy the requested size, and thus the range of indexes of nodes suitable for the allocation. Then, these nodes are scanned in order to search for a free one. When a free node is found, the allocation operation tries to (i) occupy this node by updating the occupancy bits in the status bit-mask of that node and checking if the status bits are still set to zero (step 1 in Figure 2), and (ii) propagate the information about the occupancy up to the ancestor node belonging to the lowest level by traversing the nodes along the path and updating their states. This traversal is required to set left or right (depending on what branch we are backward traversing) occupancy bits and, if necessary, clean the relative coalescent bits of the ancestor nodes so as to reflect that some sub-tree has become partially occupied (e.g. steps 2 and 3 in Figure 2). If the lowest-level node is reached and updated along the backward traversal, the node originally targeted can be considered as correctly taken. Conversely, if a node has the occupied bit (concurrently) set, which makes the CAS instruction fail to update its state, the allocation needs to be aborted and retried on a different node. The same is true if the CAS instruction fails while backward traversing and updating any node in the tree. In this case, the algorithm moves to the next candidate node by skipping the whole sub-tree relative to the node causing the failure.

b) Memory Deallocation Algorithm: A memory deallocation operation is composed by three phases. In the first phase, the ancestors of the node to be released are marked as coalescent (by setting the appropriate coalescent bit as shown in Figure 3), in order to notify that a free operation is in place along the corresponding path of the tree. If along this path a buddy is detected as occupied by other allocations and not coalescent, the climb is early stopped, since the corresponding sub-tree cannot be considered free—this is the case of Figure 3, where the right subtree of the node below the upper-bound is already fragmented. In the next phase, the node to be released is marked as free by simply writing zero on its status bits (step 1 of Figure 4). During the last phase, all the nodes previously marked as coalescent are updated again to notify that the sub-tree involving the just released node is actually free—therefore it can serve again memory requests. This is achieved by cleaning the coalescent and the occupancy bits of the traversed nodes (see steps 2 and 3 of Figure 4). The climb towards the lowest level can be stopped earlier if a node with the coalescent bit reset has been met because of concurrent deallocation/allocation operations.

Beyond providing non-blocking capabilities while allocating or releasing memory, our buddy system carries out memory fragmenting and merging operations implicitly and still in a non-blocking fashion.

III. EXPERIMENTAL RESULTS

We have compared our solution with the buddy system from version 3.2 of the Linux kernel, which is based on a multi-list data structure, and exploits spin-locks for handling concurrency. For the assessment we used 2 benchmarks taken from the literature, namely Linux scalability [6] and Thread Test [7]. 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. To test with the Linux kernel-level allocator, we developed Linux external modules that exploit the `__get_free_pages` and `free_pages` kernel services. We have configured our allocator to manage the same amount and granularity of memory as the Linux buddy system. The data in Table I refer to tests executed when targeting allocations/deallocations of 128KB chunks with 32 threads. Here, we see that our non-blocking version has performance gain of the order of 71% for Thread Test and 84% for Linux Scalability.

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 of the 23rd Workshop on Principles of Advanced and Distributed Simulation, PADS*. IEEE, 2009, pp. 45–53.
[3] 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.
[4] M. Gorman, *Understanding the Linux Virtual Memory Manager*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2004.
[5] R. Marotta, M. Ianni, A. Pellegrini, A. Scarselli, and F. Quaglia, "A non-blocking buddy system for scalable memory allocation on multi-core machines," *CoRR*, vol. abs/1804.03436, 2018.
[6] C. Lever and D. Boreham, "Malloc() Performance in a Multithreaded Linux Environment," 2000.
[7] 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.