Brief Announcement: Concurrent Fixed-Size Allocation and Free in Constant Time

Guy E. Blelloch
Carnegie Mellon University, Pittsburgh, PA, USA
guyb@cs.cmu.edu

Yuanhao Wei
Carnegie Mellon University, Pittsburgh, PA, USA
yuanhao1@cs.cmu.edu

Abstract

We describe an algorithm for supporting allocation and free for *fixed-sized* blocks, for p asynchronous processors, with $O(1)$ worst-case time per operation, $\Theta(p^2)$ *additive* space overhead, and using only single-word read, write, and CAS. While many algorithms rely on having constant-time fixed-size `allocate` and `free`, we present the first implementation of these two operations that is constant time with reasonable space overhead.

2012 ACM Subject Classification Computing methodologies → Concurrent algorithms

Keywords and phrases malloc, free, fixed-size, concurrent, constant time

Digital Object Identifier 10.4230/LIPIcs.DISC.2020.51

Related Version A full version of the paper is available at `https://arxiv.org/abs/2008.04296`.

Funding This work was supported in part by NSF grants CCF-1901381, CCF-1910030, and CCF-1919223. Yuanhao Wei was also supported by a NSERC PGS-D Scholarship.

1 Introduction

Dynamic memory allocation is an important problem that plays a role in many data structures. Although some data structures require variable sized memory blocks, many are built on fixed sized blocks, including linked lists and trees.

Our goal is to efficiently solve the dynamic memory allocation problem for fixed sized memory blocks in a concurrent setting. An `allocate`() returns a reference to a block, and a `free`(p) takes a reference p to an block. We say a block is *live* at some point in the execution history if the last operation that used it (either a `free` that took it, or `allocate` that returned it) was an `allocate`. Otherwise the block is *available*. As would be expected, an `allocate` must transition the returned block from available to live (i.e., the operation cannot return a block that is already live), and a `free`(p) must be applied to a live block p (i.e., the operation cannot free a block that is already available). In our setting, processes run asynchronously and may be delayed arbitrarily.

We describe a concurrent linearizable implementation of the fixed-sized memory allocation problem with the following properties.

▶ **Result 1** (Fixed-sized Allocate/Free). *Given m as the maximum number of live blocks, on p processes, we can support linearizable* **allocate** *and* **free** *for fixed sized blocks of $k \geq 2$ words, with*

1. *references that are just pointers to the block (i.e., memory addresses),*
2. *$O(1)$ worst-case time for each operation,*
3. *$k(m + \Theta(p^2))$ space,*
4. *single-word (at least pointer-width) read, write, and CAS.*

Our notion of time/space complexity includes both local and shared instructions/words. Limiting ourselves to pointer-width atomic instructions means that we do not use unbounded sequence numbers or hide bits in pointers. Achieving the last property of Result 1 requires using a recent result on implementing LL/SC from pointer-width CAS objects [3].

We are not addressing the problem of supporting an unbounded number of allocated cells. Indeed this would seem to be impossible without some assumption of how the operating system gives out large chunks of memory to allocate from. In our algorithm, the memory allocator could potentially read from a memory block that is live, but it works correctly regardless of what the user writes into the memory blocks. We believe these kinds of accesses are reasonable.

There has been recent progress in designing memory reclamation algorithms that are wait-free [9, 2, 7, 1]. With wait-free memory reclamation, the memory allocation part becomes the new limiting factor. While there has been work on lock-free memory allocation [6, 5, 8], the only work that we know of that is wait-free is by Aghazadeh and Woelfel [1]. Aghazadeh and Woelfel's `GetFree` and `Release` operations can be used to implement `allocate` and `free` in constant time, but their algorithm requires $\Omega(mp^3)$ space, making it impractical in many applications. We guarantee the same time complexity and wait-freedom while using significantly less space. While our algorithm is mostly a theoretical contribution, the constants are small and we believe it will be fast in practice as well.

2 Algorithm Overview

Our algorithm is fairly simple. For some constant $l \in \Theta(p)$, our data structure consists of local private pools that each hold $\Theta(l)$ blocks and a shared pool that maintains a stack of *batches*, each containing l blocks. The high level idea of maintaining separate private and shared pools is widely used and has been shown to be fast in practice [6, 5, 8]. In the common case, most calls to `allocate` and `free` will be handled directly by the private pools. Batches are transferred between shared and private pools occasionally to make sure there are not too many or too few free blocks in each private pool. Having too many private blocks weakens the bound in Item 3 of Result 1 because these blocks cannot be allocated by other processes. Pushing to and popping from the shared stack takes $O(p)$ time which is fairly expensive. To amortize this cost, push and pop can be broken up into p steps of $O(1)$ time each and performed across multiple calls to `allocate` or `free`. In the full version of our paper, we show how to manage the private pools so that there is at most one ongoing push or pop per process. For our shared pool, we start with Fatourou and Kallimanis's P-SIM stack [4] and modify it to achieve the following bounds:

▶ **Result 2** (Shared Stack). *Given a concurrent allocator satisfying Result 1 with parameter $k \geq 2$, assuming that reading from a memory block that has already been freed returns an arbitrary value, on p processes, we can support linearizable* **push** *and* **pop** *with*

1. *$O(p)$ worst-case time for each operation,*
2. *at most $2p$ calls to* **allocate** *and $2p$ calls to* **free** *in each operation,*
3. *$Mk + \Theta(p^2k)$ space (where M is the number of nodes in the stack),*
4. *single-word (at least pointer-width) read, write, and CAS.*

At first glance, it may seem circular that Results 1 and 2 are used to implement each other. However, this is the key trick in our algorithm. We observe that it is safe for the data structures in the shared pool to allocate memory from the same private pools as the user. Special care is needed to ensure that the private pools always have enough blocks to service both the user and the shared pool. For this to work, Property 2 of Result 2 is crucial.

In the P-SIM stack, push and pop operations help each other to ensure that each operation completes within $O(p)$ time (Property 1). The P-SIM stack can be modified to satisfy Property 4 using a recent simulation of LL/SC from CAS [3]. While there has been a lot of work on simulating LL/SC from CAS, [3] is the only one we are aware of that maintains the other properties in Result 2. To satisfy Properties 2 and 3, we add memory management while ensuring that each push or pop calls `free` at most $2p$ times. In the P-SIM stack, each process performs push and pop operations on its local copy of the shared stack, and then tries to set its local copy as the new shared stack using an SC. We modify this algorithm so that after each successful SC, the process frees all the nodes that it locally popped, and after each unsuccessful SC, the process frees all the nodes that it locally pushed. This modification sounds straightforward but it has a complication. It is possible for a process working on a outdated copy of a shared stack to read a node that is already freed by this approach. In most settings, accessing freed memory is not allowed, however, these accesses are reasonable in our setting because they are internal to our memory allocator. Whenever a node is popped off the shared stack and freed, the memory is not freed back to the operating system, instead it is made available to the user. Accessing memory that has been allocated to the user may return an arbitrary value, so such accesses are still dangerous. We protect against this by performing VL after every potentially dangerous access. If the VL returns true, then there has not been an SC on the shared stack since the process's last LL, so the process is working on an up-to-date view of the shared stack. This means that the earlier memory access was safe. If the VL returns false, then the process's subsequent SC is guaranteed to fail. In this case, the process restarts its operation and frees any nodes that it locally pushed. With these modifications, the P-SIM stack satisfies all the properties in Result 2.

References

1 Zahra Aghazadeh and Philipp Woelfel. Upper bounds for boundless tagging with bounded objects. In *International Symposium on Distributed Computing (DISC)*, pages 442–457, 2016.

2 Guy E. Blelloch and Yuanhao Wei. Concurrent reference counting and resource management in wait-free constant time, 2020. `arXiv:2002.07053`.

3 Guy E Blelloch and Yuanhao Wei. LL/SC and atomic copy: Constant time, space efficient implementations using only pointer-width CAS. In *International Symposium on Distributed Computing (DISC)*, 2020.

4 Panagiota Fatourou and Nikolaos D Kallimanis. A highly-efficient wait-free universal construction. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 325–334, 2011.

5 Anders Gidenstam, Marina Papatriantafilou, and Philippas Tsigas. NBmalloc: Allocating memory in a lock-free manner. *Algorithmica*, 58(2):304–338, 2010.

6 Maged M Michael. Scalable lock-free dynamic memory allocation. In *Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, pages 35–46, 2004.

7 Ruslan Nikolaev and Binoy Ravindran. Universal wait-free memory reclamation. In *ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*, page 130–143, 2020.

8 Sangmin Seo, Junghyun Kim, and Jaejin Lee. SFMalloc: A lock-free and mostly synchronization-free dynamic memory allocator for manycores. In *International Conference on Parallel Architecture and Compilation Techniques (PACT)*, pages 253–263, 2011.

9 Håkan Sundell. Wait-free reference counting and memory management. In *International Parallel and Distributed Processing Symposium (IPDPS)*, 2005.