POSTER: Understanding and Optimizing Persistent Memory Allocation

Wentao Cai, Haosen Wen, H. Alan Beadle, Mohammad Hedayati, Michael L. Scott
Computer Science Department
University of Rochester
Rochester, NY, USA
{wcai6,hwen5,hbeadle,hedayati,scott}@cs.rochester.edu

Abstract

The proliferation of fast, dense, byte-addressable nonvolatile memory suggests the possibility of keeping data in pointer-rich "in-memory" format across program runs and even crashes. For full generality, such data requires dynamic memory allocation. Toward this end, we introduce *recoverability*, a correctness criterion for persistent allocators, together with a nonblocking allocator, *Ralloc*, that satisfies this criterion. Ralloc is based on *LRMalloc* [8], with three key innovations. First, we persist just enough information during normal operation to permit reconstruction of the heap after a full-system crash. Our reconstruction mechanism performs garbage collection (GC) to identify and remedy any failure-induced memory leaks. Second, in support of GC, we introduce the notion of *filter functions*, which identify the locations of pointers within persistent blocks. Third, to allow persistent regions to be mapped at an arbitrary address, we employ the position-independent pointer representation of Chen et al. [4], both in data and in allocator metadata.

Experiments show that Ralloc provides scalable performance competitive to that of both *Makalu* [2], the state-of-the-art lock-based persistent allocator, and the best transient allocators (e.g., *JEMalloc* [5]).

CCS Concepts • **Software and its engineering → Allocation / deallocation strategies**; • **Hardware →** *Nonvolatile memory*; • **Computing methodologies →** *Shared memory algorithms.*

1 Context and Challenges

The past few years have seen a flurry of work on *persistent data structures* designed to ensure that information kept in nonvolatile memory will remain consistent in the wake of a crash. While one could in principle insist that allocation and deallocation of memory blocks be integrated into the failure-atomic operations performed on a persistent structure, this introduces nontrivial dependences among otherwise independent structures that share the same allocator. It also imposes a level of consistency (typically *durable linearizability* [7]) that is arguably unnecessary for the allocator: we do not in general care whether calls to malloc and free linearize so long as no block is ever used for two purposes simultaneously or is permanently leaked.

Leaks may arise if a crash occurs between allocating a block and attaching it persistently to the data structure—or between detaching it and deallocating it. A possible solution, exemplified by Intel's PMDK [11], is to provide malloc-to and free-from operations; these atomically and persistently allocate a block and attach it to the structure at a specified address, or break the last persistent pointer and return the block to the free list. An alternative, exemplified by HPE's *Makalu* [2], supplements a standard malloc/free interface with post-crash garbage collection to recover any blocks that might otherwise have leaked.

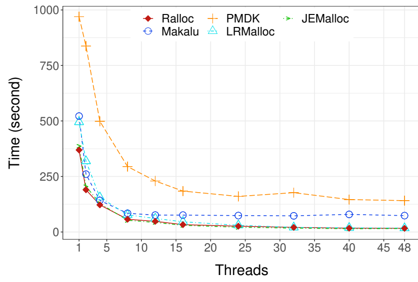
Informally, we say an allocator is *recoverable* if it ensures that, in the wake of post-crash recovery, the metadata of the allocator will indicate that all and only the "in use" blocks are allocated. In a malloc/free allocator with GC, "in use" blocks are those *reachable* from a specified set of *persistent roots.* Interestingly, given application-facilitated tracing, almost any correct, transient memory allocator can be made recoverable under a full-system-crash failure model: in the wake of a crash, a fresh copy of the allocator is reinitialized to reflect the enumerated set of in-use blocks. Very little in the way of allocator metadata needs to be persisted.

If long-lived data are to be kept "in memory" across program runs, it seems attractive to allow them to be shared, concurrently, by independently developed programs [6]. This raises the prospect of independent process failures; it motivates the use of *nonblocking* data structures—and a nonblocking allocator—to avoid the logging and on-line recovery required by lock-based code. (In the absence of failures, nonblocking algorithms also protect against performance anomalies caused by inopportune preemption.)

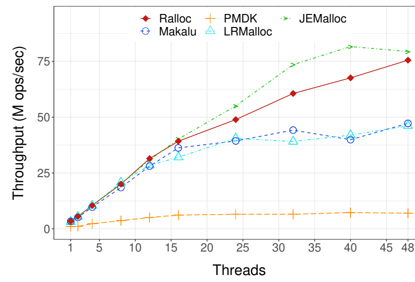
Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

PPoPP '20, February 22–26, 2020, San Diego, CA, USA
© 2020 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-6818-6/20/02.
https://doi.org/10.1145/3332466.3374502

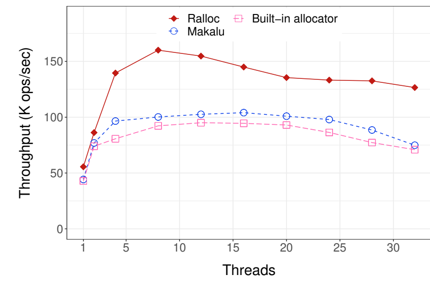
This work was supported in part by NSF grants CCF-1422649, CCF-1717712, and CNS-1900803, and by a Google Faculty Research award.



(a) Threadtest (lower is better)



(b) Larson (higher is better)



(c) Memcached (higher is better)

2 Ralloc

Our allocator, *Ralloc*, is based on the (transient) *LRMalloc* [8], which is in turn derived from Michael's nonblocking allocator [9]. Most metadata is transient and is reconstructed after a crash. Thread-local caching allows most allocator operations to be fulfilled without any synchronization. Empty *superblocks* (contiguous collections of blocks) are kept on a free list rather than being unmapped; this change allows Ralloc to outperform LRMalloc, despite persistence.

Many tracing garbage collectors assume that each block is self descriptive, at least with regard to size. Ralloc, by contrast, relies on the fact that all blocks in a given superblock are of identical size. It persists this size whenever allocating a new superblock, which is rare; in a typical operation, nothing needs to be explicitly persisted.

In a type-safe language, Ralloc could (in principle) rely on type information provided by the compiler to enumerate reachable blocks. To support unsafe languages like C/C++, Ralloc relies by default on *conservative* collection [3]. Tracing begins in a set of *persistent roots*, which must be exported by the application, but further blocks are deemed reachable if their starting addresses correspond to a word-aligned 64-bit value that is itself in a reachable block.

The problem, of course, with conservative collection is the possibility of memory leaks caused by false positives during tracing. To mitigate this problem, Ralloc allows programmers to specialize a *filter function* that enumerates internal pointers for a given type of block. As an example, the following defines a filter function for binary tree nodes:

```
1 Class TreeNode
2     ... // content fields
3     left, right : TreeNode* ;
4 Func filter(TreeNode* ptr)
5     visit(ptr→left) ;
6     visit(ptr→right) ;
```

Here visit recursively invokes the filter function of its target.

In keeping with Makalu but in contrast to PMDK, Ralloc provides a traditional malloc/free interface, rather than malloc-to/free-from. In addition to making it easier to port existing application code, this interface relieves the programmer of the need to keep track, in persistent memory, of nodes that have been allocated but not yet attached to the main data structure—perhaps because of speculation, or because they are still being initialized.

To allow persistent structures to be mapped at different virtual addresses in different processes and across multiple executions, we implement *position-independent data* using location-relative "off-holder" pointers [4]. These are substantially more flexible than the once-and-for-always fixed addresses of Makalu and more space efficient than the 128-bit base-plus-offset pointers of PMDK. Each off-holder specifies the distance between its own location and that of the pointed-at block.

3 Experiments

We evaluated the performance of Ralloc on an Intel server using the well known Threadtest and Larson benchmarks [1] and persistent *Memcached* from the WHISPER suite [10]. As shown above, Ralloc consistently matches or outperforms known alternatives.

References

[1] E. D. Berger, K. S. McKinley, R. D. Blumofe, and P. R. Wilson. Hoard: A scalable memory allocator for multithreaded applications. In *ASPLOS*, Cambridge, MA, Nov. 2000.

[2] K. Bhandari, D. R. Chakrabarti, and H.-J. Boehm. Makalu: Fast recoverable allocation of non-volatile memory. In *OOPSLA*, Amsterdam, The Netherlands, Oct. 2016.

[3] H.-J. Boehm and M. Weiser. Garbage collection in an uncooperative environment. *Software: Practice and Experience*, 18(9):807–820, Sept. 1988.

[4] G. Chen, L. Zhang, R. Budhiraja, X. Shen, and Y. Wu. Efficient support of position independence on non-volatile memory. In *MICRO*, Cambridge, MA, Oct. 2017.

[5] J. Evans. A scalable concurrent malloc (3) implementation for FreeBSD. In *BSDCan Conf.*, Ottawa, ON, Canada, May 2006.

[6] M. Hedayati, S. Gravani, E. Johnson, J. Criswell, M. L. Scott, K. Shen, and M. Marty. Hodor: Intra-process isolation for high-throughput data plane libraries. In *ATC*, Renton, WA, July 2019.

[7] J. Izraelevitz, H. Mendes, and M. L. Scott. Linearizability of persistent memory objects under a full-system-crash failure model. In *DISC*, Paris, France, Sept. 2016.

[8] R. Leite and R. Rocha. LRMalloc: A Modern and Competitive Lock-Free Dynamic Memory Allocator. In *VECPAR*, São Pedro, Brazil, Sept. 2018.

[9] M. M. Michael. Scalable lock-free dynamic memory allocation. In *PLDI*, Washington DC, USA, June 2004.

[10] S. Nalli, S. Haria, M. D. Hill, M. M. Swift, H. Volos, and K. Keeton. An analysis of persistent memory use with WHISPER. In *ASPLOS*, Xi'an, China, 2017.

[11] A. Rudoff and M. Slusarz. Persistent memory development kit, Sept. 2014. http://pmem.io/pmdk/.