Understanding and Optimizing Persistent Memory Allocation

Wentao Cai
Haosen Wen
University of Rochester
Rochester, NY, USA
{wcai6,hwen5}@cs.rochester.edu

H. Alan Beadle
Chris Kjellqvist
University of Rochester
Rochester, NY, USA
{hbeadle,ckjellqv}@cs.rochester.edu

Mohammad Hedayati
Michael L. Scott
University of Rochester
Rochester, NY, USA
{hedayati,scott}@cs.rochester.edu

Abstract

The proliferation of fast, dense, byte-addressable nonvolatile memory suggests that data might be kept in pointer-rich "in-memory" format across program runs and even process and system crashes. For full generality, such data requires dynamic memory allocation, and while the allocator could in principle be "rolled into" each data structure, it is desirable to make it a separate abstraction.

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 the *LRMalloc* of Leite and Rocha, with four key innovations: First, we persist just enough information during normal operation to permit a garbage collection (GC) pass to correctly reconstruct the heap in the wake of a full-system crash. Second, we introduce the notion of *filter functions*, which identify the locations of pointers within persistent blocks to mitigate the limitations of conservative GC. Third, we reorganize the layout of the heap to facilitate the incremental allocation of physical space. Fourth, we employ position-independent (offset-based) pointers to allow persistent regions to be mapped at an arbitrary address.

Experiments show Ralloc to be performance-competitive with both *Makalu*, the state-of-the-art lock-based persistent allocator, and such transient allocators as LRMalloc and JEMalloc. In particular, reliance on GC and offline metadata reconstruction allows Ralloc to pay almost nothing for persistence during normal operation.

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

ISMM '20, June 16, 2020, London, UK

ACM ISBN 978-1-4503-7566-5/20/06…$15.00
https://doi.org/10.1145/3381898.3397212

Keywords: dynamic memory allocation, nonvolatile memory, lock freedom, garbage collection, persistent pointer

ACM Reference Format:
Wentao Cai, Haosen Wen, H. Alan Beadle, Chris Kjellqvist, Mohammad Hedayati, and Michael L. Scott. 2020. Understanding and Optimizing Persistent Memory Allocation. In *Proceedings of the 2020 ACM SIGPLAN International Symposium on Memory Management (ISMM '20), June 16, 2020, London, UK*. ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3381898.3397212

1 Introduction

Byte-addressable nonvolatile memory (*NVM*) offers significantly higher capacity and lower energy consumption than DRAM, with a latency penalty of less than an order of magnitude. Intriguingly, NVM also raises the possibility that applications might access persistent data directly with load and store instructions, rather than serializing updates through a block-structured file system. Taking advantage of persistence, however, is not a trivial exercise. If data is to be recovered after a full-system crash, the contents of NVM must always represent a consistent logical state—ideally one that actually arose during recent pre-crash execution [24]. Ensuring such consistency generally requires instrumentation with explicit write-back and fence instructions, to avoid the possibility that updated values may still reside only in the (transient) cache when data that depend upon them have already been written back. To avoid the need to instrument by hand, various groups have developed persistent versions of popular data structures [17, 37, 54, 55] as well as more general techniques to add failure atomicity to code based on locks [6, 23, 30], transactions [1, 10, 14, 39, 50], or both [41].

One could in principle insist that memory management be integrated into the failure-atomic operations performed on persistent structures, but this has the disadvantage of introducing dependences among otherwise independent structures that share the same allocator. It also imposes a level of consistency (typically *durable linearizability* [24]) 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.

As in work on transactional memory [21], it is desirable to provide `malloc` and `free` as primitives to the authors of persistent data structures. In so doing, one must consider how

to avoid memory leaks if a crash occurs between allocating a block and attaching it persistently to the data structure—or between detaching and deallocating it. Intel's Persistent Memory Development Kit (PMDK) [41] exemplifies one possible approach, in which the allocator provides a `malloc-to` operation that allocates a block and, atomically, attaches it persistently at a specified address. A similar `free-from` operation breaks the last persistent pointer to a block and, atomically, returns it to the free list. HPE's Makalu [3] exemplifies an alternative approach, in which a traditional `malloc`/`free` interface is supplemented with post-crash garbage collection to recover any blocks that might have leaked.

We adopt Makalu's approach in our work. In addition to making it easier to port existing application code, the traditional interface allows us to eliminate write-back and fence instructions in allocator code, and frees 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.

As a correctness criterion for a persistent allocator, we introduce the notion of *recoverability*. Informally, we say an allocator is recoverable if, in the wake of post-crash recovery, it ensures that the metadata of the allocator will indicate that all and only the "in use" blocks are allocated. In a `malloc-to`/`free-from` allocator, "in use" blocks would be defined to be those that have (over the history of the structure, including crashes) been `malloc-to`-ed and not subsequently `free-from`-ed. In a `malloc`/`free` allocator with GC, "in use" blocks are defined to be those that are *reachable* from a specified set of *persistent roots*. In this case, the application and the allocator must agree on a tracing mechanism that enumerates the reachable blocks. In the case of conservative tracing it is conceivable that a block will appear to be reachable without ever having been allocated; by treating such blocks as "in use," we admit the possibility that a crash will leak some memory. As in prior work [4], this never compromises safety, and leaked blocks may often be recovered in subsequent collections, if values erroneously interpreted as references have changed.

Armed with this notion of correctness, we present what we believe to be the first nonblocking recoverable allocator. Our system, *Ralloc*, is based on the (transient) *LRMalloc* of Leite and Rocha [28], which is in turn derived from Michael's nonblocking allocator [32]. Like LRMalloc, Ralloc uses thread-local caching to fulfill most allocation and deallocation requests without synchronization. When it does need to synchronize, it commonly issues two *compare-and-swap* (CAS) instructions and a write-back & fence pair in `malloc` or `free`. Most metadata needed for fast operation resides only in transient memory (with no explicit writes-back required) and is reconstructed after a full-system crash. In the event of a partial crash (e.g., due to a software bug outside the allocator that takes down one of several cooperating processes), memory may be leaked on a temporary basis: it can be recovered via garbage collection in some subsequent quiescent interval.

For type-unsafe languages like C and C++, Ralloc adopts the conservative strategy of Boehm and Weiser [4]. To accelerate recovery, accommodate nonstandard pointer representations, and reduce the likelihood of erroneously unrecoverable blocks due to false positives during tracing, we introduce what we call *filter functions*—optional, user-provided routines to enumerate the pointers in a given block.

To allow persistent structures to be mapped at different virtual addresses in different processes and at different times, we use an offset-based pointer representation [8, 10] to provide fully *position-independent data*. (Specifically, each pointer stores the 64-bit signed offset of the target from the pointer itself.) By contrast, several existing systems force data to reside at the same address in all processes across all of time [3, 50]; others expand the size of each pointer to 128 bits for base-plus-offset addressing [38, 41]. The former approach introduces an intractable bin-packing problem as application needs evolve, and is incompatible with *address space layout randomization* (ASLR) [44] for security; the latter introduces space overhead and forces the use of a *wide-compare-and-swap* (WCAS) for atomic updates.

Summarizing contributions:

- (Section 3) We introduce *recoverability* as the correctness criterion for persistent allocators, eschewing unnecessary ordering among allocator operations and preserving the essential properties of conventional transient allocators for a world with persistent memory.
- (Section 4) We introduce an allocator, *Ralloc*, that is fast, nonblocking, and recoverable, and that provides a standard API. Ralloc incorporates *filter functions* to enhance the performance, generality, and accuracy of conservative garbage collection. It also incorporates offset-based smart pointers for fast position independence.
- (Section 5) We argue that Ralloc is both safe and live: it never overlaps in-use blocks, it eventually reuses freed blocks, it is lock-free, and it is recoverable.
- (Section 6) We present performance results confirming that Ralloc scales well to large numbers of threads and is performance competitive, on both allocation benchmarks and real applications, with both JEMalloc [15] and Makalu [3].

2 System Model

2.1 Hardware and Operating System

We assume that NVM is attached to the system in parallel with DRAM, and directly exposed to the operating system (OS) as byte-addressable memory. This model matches (but is not limited to) recent Intel machines, in which Optane

DIMMs are configured in so-called *App Direct* mode [25].[1] The OS, in turn, makes NVM available to applications through a *direct access* (DAX) [40] mechanism in which persistent memory segments have file system names and can be mapped directly into user address spaces. DAX employs a variant of `mmap` that bypasses the traditional buffer cache [9, 52–55]. Like traditional `mmap`-ed files, DAX consumes physical NVM on demand: a page of physical memory is consumed only when it is accessed for the first time. This feature allows the programmer to define memory segments that are large enough to accommodate future growth, without worrying about space lost to internal fragmentation.

Some DAX operations (e.g., `mmap`) have effects that are entirely transient: they are undone implicitly on a system shutdown or crash. We assume that all others are failure atomic—that is, the OS has been designed (via logging and boot-time recovery) to ensure that they appear, after recovery, to have happened in their entirety or not at all. By contrast, updates to DAX files mapped into user-level programs are ordinary memory loads and stores, filtered through volatile caches that reorder writes-back during normal operation, and that lose their contents on a full-system crash.

Applications that wish to ensure consistency after a crash must generally employ special hardware instructions to control the order in which lines are written back to NVM. On recent Intel processors [22, 40], the `clflush` instruction evicts a line from all caches in the system, writes it back to memory if dirty, and performs a store fence to ensure that no subsequent store can occur before the write-back. The `clflushopt` instruction does the same but without the store fence; `clwb` performs the write-back without necessarily evicting or fencing. The latter two instructions can be fenced explicitly with a subsequent `sfence`. In keeping with standard (if somewhat inaccurate) usage in the literature, the rest of this paper uses "flush" to indicate what will usually be a `clwb` instruction, and uses "fence" for `sfence`.

We assume that a persistent data structure must, at the very least, tolerate *full system, fail-stop* crashes, as might be caused by power loss or the failure of a critical hardware component. On such a crash, dirty data still in cache may be lost, but writes-back at cache-line granularity will never be torn, and there is no notion of Byzantine behavior.

More ambitiously, we wish to accommodate systems that share data among mutually untrusting applications with independent failure modes. This stands in contrast to previous projects, which have assumed that all threads sharing a given persistent segment are part of a single application, and are equally trusted. Recent work [19, 48] has shown that it is possible, at reasonable cost, to amplify access rights when calling into a *protected library* and to reduce those rights on return. The OS, moreover, can arrange for any thread currently executing in a protected library to finish the current operation cleanly in the event its process dies (assuming, of course, that the library itself does not contain an error). Applications that trust the library can then share data safely, without worrying that, say, a memory safety error in another application will compromise the data's integrity. Protected libraries have the potential to greatly increase performance, by allowing a thread to perform an operation on shared memory directly, rather than using interprocess communication to ask a server to perform the operation on its behalf. They introduce the need to accommodate situations in which recovery from process crashes, if any, proceeds in parallel with continued execution in other processes.

[1] Intel also supports an alternative *Memory Mode*, in which DRAM serves as a hardware-managed cache for the larger but slower Optane memory, whose persistence is ignored. We ignore this alternative in our work.

2.2 Runtime and Applications

We assume that every persistent data structure (or group of related structures) resides in a *persistent segment* that has a name in the DAX file system and can be mapped into contiguous virtual addresses in any program that wants to use it (and that has appropriate file system rights). The goal of our allocator is to manage dynamically allocated space within such segments. We assume, when a structure is quiescent (no operations active), that any useful block will be reachable from some static set of *persistent roots*, found at the beginning of the segment. We further assume that the OS allows a manager process to be associated with any segment that is shared among applications, and that it notifies this manager whenever a process sharing the segment has crashed (but the system as a whole has not). Finally, we assume that an application can tell when it is the first active user of any given segment, allowing it to perform any needed recovery from a full-system crash (if the segment was not cleanly closed) and to start any needed manager process.

For all persistent data, we assume that application code takes responsibility for *durable linearizability* [17, 24] or its buffered variant. Durable linearizability requires that data structure operations persist, in linearization order, before returning to their callers. Buffered durable linearizability relaxes this requirement to allow some completed operations to be lost on a crash, so long as the overall state of the system (after any post-crash recovery) reflects a consistent cut across the happens-before order of data structure operations. Both variants extend in a straightforward way to accommodate fail-stop crashes of a nontrivial subset of the active threads. They do not encompass cases in which a crashed thread recovers and attempts to continue execution where it last left off. New threads, however, may join the execution.

While some data structures may be designed specifically for persistence and placed in libraries, the requisite level of hand instrumentation is beyond most programmers. To facilitate more general use of persistence, several groups have developed libraries and, in some cases, compiler-based systems to provide failure atomicity for programmer-delimited blocks

of code. In some systems, these blocks are outermost lock-based critical sections, otherwise known as *failure-atomic sections* (FASEs); examples in this camp include Atlas [6], JUSTDO [23], and iDO [30]. In other systems, the atomic blocks are *transactions*, which may be speculatively executed in parallel with one another; examples in this camp include Mnemosyne [50], NV-Heaps [10], QSTM [1], and OneFile [39]. Intel's PMDK library [41] also provides a transactional interface, but solely for failure atomicity, not for synchronization among concurrently active threads.

3 Recoverability

Our allocator needs to be compatible with all the programming models described in the previous section. As described in Section 1, it must also address the possibility that a crash may occur after a block has been allocated but before it has been made reachable from any persistent root—or after it has been detached from its root but before it has been reclaimed. Rather than force these combinations to persist atomically, together, we rely on post-crash garbage collection to recover any memory that leaks. While GC-based systems require a mechanism to trace the set of in-use blocks, they have compelling advantages. Use of a standard API avoids the need to specify attachment points in calls to `malloc-to` and `free-from`; it also facilitates porting of existing code. More importantly, the garbage collector's ability to reconstruct the state of the heap after a crash avoids the ongoing cost of flushing and fencing both allocator metadata and, for nonblocking structures, the *limbo lists* used for safe memory reclamation [16, 31, 51].

We say that a persistent allocator is *recoverable* if, in the wake of a crash, it is able to bring its metadata to a state in which all and only the "in use" blocks are allocated. For applications using the `malloc-to`/`free-from` API, "in use" blocks can be defined to be those that have (over the history of the structure, including crashes) been `malloc-to`-ed and not subsequently `free-from`-ed. In a `malloc`/`free` allocator with GC, we define "in use" blocks to be those that are reachable from the persistent roots. The notion of reachability, in turn, requires a mechanism to identify the pointers in each node of the data structure, so that nodes can be traced recursively. If the identification mechanism is conservative, then some blocks that were never actually allocated prior to a crash may be considered to be "in use" after recovery.

We observe that, given an appropriate tracing mechanism, almost any correct, transient memory allocator can be made recoverable under a full-system-crash failure model. During normal operation, no block will be leaked or used for more than one purpose simultaneously; in the wake of a crash, a fresh copy of the allocator can be reinitialized to reflect the enumerated set of in-use blocks. (In a type-safe language, the reinitialization process may also perform compaction. This is not possible with conservative collection.) Very little in the way of allocator metadata needs to be saved consistently to NVM. This observation transforms the central question of persistent allocation from "how do we persist our `malloc` and `free` operations?" to "how do we trace our data structures during recovery?"

But not all allocators are created equal. There are compelling reasons, we believe, why a persistent allocator should employ *nonblocking* techniques. First, a blocking allocator inherently compromises the progress of any otherwise nonblocking data structure that relies on it for memory management. Second, nonblocking algorithms dramatically simplify the task of post-crash recovery, since execution can continue from any reachable state of the structure (and the allocator). Third, even if cross-application sharing employs a protected library that arranges to complete all in-flight operations in a dying process, the problem of priority inversion suggests that a thread should never have to wait for progress in a different protection domain.

Among existing transient allocators, the first nonblocking implementation is due to Maged Michael [32]. It makes heavy use of the CAS instruction in allocation and deallocation and is noticeably slower than the fastest lock-based allocators. The more recent LRMalloc of Leite and Rocha [28] uses thread caching to reduce the use of CAS on its "fast path," and makes allocations and deallocations mostly synchronization free. Other lock-free allocators include NBmalloc [18] and SFMalloc [43]. Due to the complexity of their internal data structures, these appear much harder to adapt to persistence; our own work is based on LRMalloc.

Despite the development of nonblocking allocators, fast, nonblocking *concurrent* (online) collection remains an open research problem. We adopt the simplifying assumption that crashes are rare (mean times to failure on the order of hours to months) and that a blocking approach to GC will be acceptable in practice. It is clearly so in the wake of a full-system crash. In the event of partial (single-process) failures, memory may leak temporarily. If the allocator identifies a low-memory situation and knows that one or more processes have crashed since GC was last performed, it can initiate a stop-the-world collection.

4 Ralloc

From LRMalloc, Ralloc inherits the notion of thread-local caches of free blocks; allocations and deallocations move blocks from and to these caches in most cases, avoiding synchronization. The rare cases that require synchronization typically entail only two CAS instructions.

In adapting LRMalloc to persistence, we introduce four principal innovations:

1. We rely on the fact that all blocks in a given *superblock* (major segment of the heap) are of identical size to avoid the need to persistently maintain a size field in blocks. Instead, we persist the common size during superblock

```
1 Class Ralloc
2     Function init(string path, int size) : bool
          // create or remap heap in path
          // size of superblock region
          // return true if heap files exist
3     Function recover() : bool
          // issue offline GC if it is restart
          // return true if heap was dirty
4     Function close() : void
          // give back cached blocks and flush heap
          // set heap as not dirty
5     Function malloc(int size) : void*
          // allocate size bytes
          // return address of allocated block
6     Function free(void* ptr) : void
          // deallocate ptr
7     Function setRoot(void* ptr, int i) : void
          // set ptr to be root i
8     Function getRoot<T>(int i) : T*
          // update root type info
          // return address of root i
```

Figure 1. API for Ralloc.

allocation, which is rare. In the typical `malloc` operation, nothing needs to be written back to memory explicitly.

2. To improve the speed, accuracy, and generality of conservative garbage collection, we introduce the notion of *filter functions*. These serve to enumerate the references found in a given block, for use during trace-based collection. In the absence of a user-provided filter function, we conservatively assume that every 64-bit aligned bit pattern is a potential reference.
3. We reorganize LRMalloc's data into three respectively contiguous regions, and utilize the major (superblock) region in increasing order of virtual address as demand increases. This strategy ensures that physical pages will be allocated lazily by the operating system.
4. To allow a persistent heap to be mapped at different addresses in different applications (or instances of the same application over time), we use an offset-based pointer representation [8, 10] for Ralloc's metadata references, and encourage applications to do the same for data structure pointers. The result can aptly be described as *position-independent data*. In our code base, offsets are implemented as C++ smart pointers.

4.1 API

The API of Ralloc is shown in Figure 1. Function `init()` must be called to initialize Ralloc prior to using it. This function checks the persistent heap referenced by the parameter `path` to determine whether this is a fresh start, a clean restart without unaddressed failure, or a dirty restart from a crash. A fresh start will create the persistent heap on NVM, map it in DAX mode, initialize the metadata, and return `false`. A clean restart will remap persistent heap to the address space and also return `false`. A dirty restart will re-map the heap and return `true`, indicating that recovery is needed. If the application receives `true` from `init()`, it will need to call `recover()` (after calling `getRoot<T>()`—see below) to invoke the offline recovery routine and reconstruct metadata.

It is the programmer's responsibility to provide a sufficiently large `size` to `init()`. If space runs out, calls to `malloc()` will fail (return `null`). Resizing currently requires an allocator restart and an `init()` call with a larger size. As a practical matter, resizing only changes the first word of the superblock region and calls `mmap` with a larger size; no data rearrangement is required.

At the end of application execution, `close()` should be called to gracefully exit the allocator. In the process, any blocks held in thread-local caches will be returned to their superblocks, and the persistent heap will be written back to NVM for fast restart.

The functions used for allocation and deallocation are similar to the traditional `malloc()` and `free()`. Allocation and deallocation requests are segregated by *size class*. Most requests are fulfilled from thread-local caches of blocks of each size class, avoiding synchronization. If the relevant cache is empty, Ralloc either fetches a partially used *superblock* (a chunk of blocks in the same size) from the *partial list* of the size class, or fetches a free superblock from the *superblock free list*. Both lists are accessible to all threads. All free blocks in a fetched superblock will be pushed into the thread-local cache. If the cache becomes too full as a result of deallocation, blocks will be returned to their superblocks, which may then appear in a partial list. Additional implementation details appear in Section 4.4.

In support of garbage collection, Ralloc maintains a set of *persistent roots* for data structures found in the heap. These serve as the starting points for tracing. The `setRoot()` and `getRoot<T>()` routines are used to store and retrieve these roots, respectively. In C++, `getRoot<T>()` is generic in the type `T` of the root; as a side effect, it associates with the root a pointer to the `T` specialization (if any) of the GC filter function (Section 4.5.1), avoiding the need for position-independent function pointers. When `init()` returns `true`, the application should call `getRoot<T>()` for each persistent root before it calls `recover()`.

4.2 Data Structures

A Ralloc heap comprises a *superblock region*, a *descriptor region*, and a *metadata region*, all of which lie in NVM (Figure 2). Only the fields shown in bold are flushed and fenced explicitly online. (All fields are eventually written back implicitly, of course, allowing quick restart after a clean shutdown.) The three regions are respectively mapped into the address space of the application. The superblock region, which is by far the largest of the three, begins with an indication of its

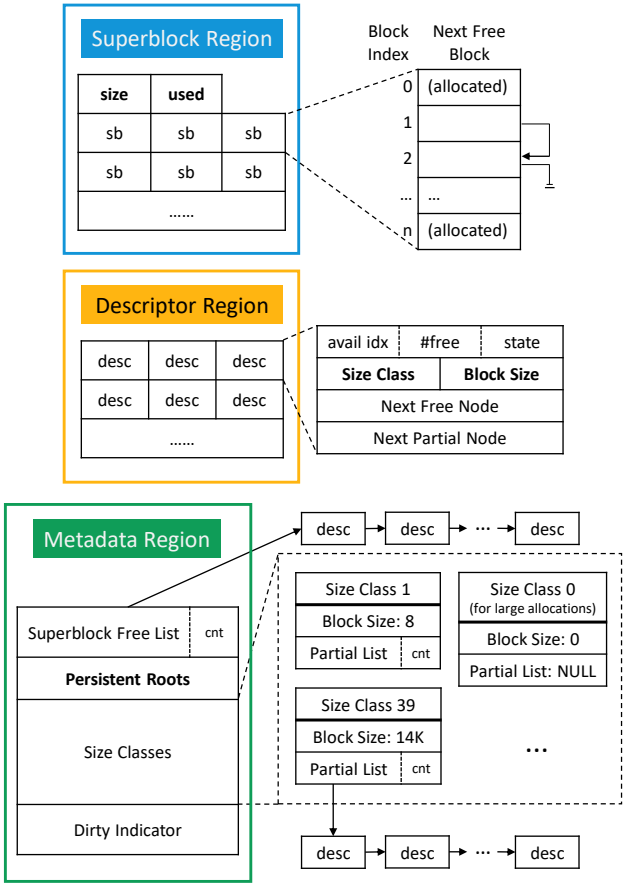


Figure 2. Superblock, Descriptor, and Metadata Regions. Variables written back explicitly online are **bold**.

maximum size, which is set at initialization time and never changed. A second word indicates the size of the prefix that is currently in use. The descriptor region is always allocated at its maximum size of *size*/1024 (a superblock is 64 KB whereas a descriptor is 64 B). The metadata region has a fixed size, dependent on the number of size classes, but not on the size of the heap.

The superblock region holds the actual data of the heap. After the initial *size* and *used* words, it holds an array of *superblocks*, each of which comprises an array of blocks. All blocks in a given superblock belong to the same size class. If a block is free (not in use), its first word contains a pointer to the next free block, if any, in the same superblock.

A *descriptor* describes a superblock, and is the locus of synchronization on that superblock. Each descriptor is 32 B in size, padded out to a 64 B cache line. Within a given heap, the *i*-th descriptor corresponds to the *i*-th superblock, allowing either to be found using simple bit manipulation given the location of the other. Each descriptor contains five fields: a 64 b *anchor*, a *size class* index, a *block size*, and two optional pointers to form the *superblock free list* and a *partial list*. The anchor, which is updated atomically with CAS, indicates the index of the first block on the block free list, the number of free blocks, and the *state* of the corresponding superblock. The state is one of `EMPTY`, `PARTIAL`, or `FULL`, meaning the superblock is entirely free, partially allocated, or fully allocated. The *size class* field indicates which of several standard sizes is being used for blocks in the superblock, or 0 if the superblock comprises a single block that is larger than any standard size. The *block size* field indicates the size of each block in this superblock, either fetched from a size class or set to the actual size of a large block during allocation. When the superblock of this descriptor is in the superblock free list or a partial list, one of the descriptor's pointer fields is set to the next node in the list.

The *size class* field (and also *block size* if it is a large block) has to be persisted before a superblock is used for allocation, because Ralloc needs the size information of every reachable block to recover metadata after a crash.

The metadata region holds the portion of Ralloc's metadata, other than descriptors, that is needed on a clean restart. Unlike LRMalloc, which always calls `munmap()` to give empty superblocks back to the OS, Ralloc implements a *superblock free list*. This list is a lock-free LIFO list (a Treiber stack [46]) of descriptors, linked through their *next free node* fields. Given the 1-to-1 correspondence between superblocks and descriptors, Ralloc finds a free superblock easily given a pointer to its descriptor.

Persistent roots point to the external entry points of persistent data structures in the superblock region. They comprise the starting points for tracing during garbage collection: only blocks identified as potentially reachable from the roots will be preserved; all other blocks will be identified as not in use. In our current implementation, a metadata region contains 1024 roots; applications can initialize as many of these as required for a given set of data structures.

Our current implementation supports 39 different *size classes*, supporting blocks that range from 8 to 14 K bytes [28]. A 40th class (number 0) supports blocks that are larger than those of any standard class. Each superblock holds blocks of exactly one class. Each size class metadata record contains the block size and a *partial list*; elements on this LIFO lists are descriptors (linked through their *next partial node* fields) for partially filled superblocks whose blocks are of the given size class. The heads of both the partial lists and the superblock free list have 34 bits devoted to a counter (a benefit of the persistent pointers discussed in Section 4.6 below) to avoid the *ABA problem* [42, Sec. 2.3.1].

Ralloc uses a *dirty indicator*, implemented as a "robust" `pthread_mutex_t` [45], to indicate whether it is necessary to perform recovery before using the allocator. Every time a process starts or restarts a Ralloc heap, Ralloc tries to lock the mutex. If it fails with error code `EOWNERDEAD`, meaning that the previously owning thread terminated with the mutex locked, then the allocator was not cleanly shut down.

During a normal exit, after all metadata has been written back to NVM, the mutex is unlocked. Since a robust `pthread_mutex_t` records failure for only a single application, we will need to extend this mechanism to accommodate independent failures of processes sharing a heap; more on this in Section 4.5.2.

In addition to its three persistent regions, Ralloc maintains transient thread-local caches of blocks of each size class. In the event of a crash, all record of blocks held in thread-local caches will be lost, and must be recovered via garbage collection. On a clean shutdown, the thread-local caches are naturally empty.

Superblocks, descriptors, size classes, partial lists, and thread-local caches are all inherited from LRMalloc. Ralloc reorganizes them into three contiguous regions; adds persistent roots, the superblock free list, and the dirty indicator; links descriptors rather than superblocks in partial lists; and persists the necessary fields.

4.3 Persistent Region Management

In order to limit the length of the superblock free list, initially only 1 GB of a heap's superblock region is included. More superblocks are made available on demand until the heap reaches the *size* limit specified in the most recent call to `init()`. Within the specified limit, Ralloc obtains more space by CAS-ing a larger number into the `used` field (with an explicit flush and fence). This update happens inside `malloc` either when no superblock is available or when a large allocation request is made. Subsequent initialization with a smaller size will trigger an exception.

4.4 Allocation and Deallocation

Allocation requests for "small" objects (14 KB or less) are segregated by size class. The appropriate thread-local cache is typically not empty, allowing the request to be fulfilled without any synchronization. An empty thread-local cache will be refilled before satisfying the allocation request. The refill operation grabs all available blocks in a partial superblock, or all blocks in a new superblock if the partial list is empty. The anchor in the corresponding descriptor is updated with a CAS to indicate the change. If a new superblock is needed but the superblock free list is empty, that list is refilled by expanding the used space of the superblock region. Our current implementation performs such expansion in 1 GB increments. We did not observe significant changes in performance with larger or smaller expansion sizes.

When a small object is being deallocated, its descriptor is found via bit manipulation. Ralloc determines its size class from the descriptor. If the thread-local cache is not full, then the freed block is simply added to the cache. Otherwise, all of the blocks in the cache are first pushed back to the block free list(s) of their respective superblock(s). A descriptor changed from `FULL` to `PARTIAL` is pushed to the partial list; a descriptor changed from `FULL` to `EMPTY` is retired and pushed to the superblock free list. A descriptor that is changed from `PARTIAL` to `EMPTY` will be retired, later, when it is fetched from the partial list.

Allocation and deallocation routines for small objects are inherited largely from LRMalloc. Given space constraints, the code is not shown here; it differs from the original mainly in the addition of flush and fence instructions needed to persist the fields shown in bold in Figure 2. Large objects see a bit more change in the code. In LRMalloc, any allocation over 14 KB is fulfilled directly by `mmap` at an arbitrary virtual address. This approach is not applicable in Ralloc because all of our allocations must lie in the same persistent segment (i.e., the superblock region). Ralloc therefore rounds the size of a large allocation up to a multiple of the superblock size (64 KB) and allocates it by expanding the used space in the superblock region. The size is persistently stored in the first corresponding descriptor. Although this approach may introduce some external memory fragmentation, we consider it acceptable if large allocations are rare. Ralloc with mostly small allocations has no external fragmentation and little internal fragmentation.

When a large block is deallocated, it is split into its constituent superblocks, which are then pushed to the superblock free list. Both allocation and deallocation, for both small and large objects, are lock-free operations. Updates to persistent fields (those shown in bold in Figure 2) are flushed and fenced to enable post-crash recovery. Other fields are reconstructed during recovery.

4.5 Recovery

Recovery employs a tracing garbage collector to identify all blocks that are reachable from the specified persistent roots. Because the sizes of all blocks are determined by their superblock (whose size field is persisted), it is easy to tell how much memory is rendered reachable by any given pointer (pointers to fields *within* a block are not supported). After GC, all metadata is reconstructed. In a bit more detail, recovery comprises the following steps:

1. Remap persistent regions to memory.
2. Initialize thread-local caches as empty.
3. Initialize empty superblock free and partial lists.
4. Set the filter function for each persistent root.
5. Trace all blocks reachable from persistent roots and put their addresses in a transient set.
6. Scan superblock region and keep only traced blocks.
7. Update each descriptor accordingly.
8. Reconstruct the partial list in each size class.
9. Reconstruct the superblock free list.
10. Flush the three persistent regions and issue a fence.

Steps 1–3 are done in `init()`. Step 4 is done when `getRoot<T>()` is called for each persistent root. Steps 5–10 are done in `recover()`. When `init()` is called, the dirty indicator (see Section 4.2) is reinitialized and set dirty until

```
1  Class Ralloc
2      ... // functions mentioned in API and metadata
3      roots : Persistent roots ;
4      rootsFunc : Functions for persistent roots ;
5      Function getRoot<T> (int i) : T*
6          rootsFunc[i] =
7          Lambda [ ](void* p, GC& gc) : void
8              gc.visit<T>(p);
9          return roots[i] ;
10 Class GC
11     visitedBlk : Set of visited blocks ;
12     pendingBlk : Stack of pending blocks to be visited ;
13     pendingFunc : Stack of functions of pending blocks ;
14     Function visit<T> (T* ptr) : void
15         if ptr ∈ superblock region and ptr ∉ visitedBlk then
16             insert ptr to visitedBlk ;
17             push ptr to pendingBlk ;
18             push Lambda [ ](void* p, GC& gc) : void
19                 gc.filter<T>(p);
20             to pendingFunc ;
21     Function filter<T> (T* ptr) : void
           // Default conservative filter function
22         get descriptor desc of ptr ;
23         read block size size from desc ; // 0 if invalid
24         for i = 0 to size−1 by alignof(pptr<>) do
25             read potential pointer curr at offset i in *ptr ;
26             visit(curr);
27     Function collect () : void
           // To get the set of reachable blocks
28         for i = 0 to max root do
29             if roots[i] ≠ NULL then
30                 rootsFunc[i](roots[i], *this) ;
31         while pendingBlk ≠ ∅ do
32             pop func from pendingFunc;
33             pop blk from pendingBlk;
34             func(blk, *this) ;
```

Figure 3. Filtered garbage collection.

a call to `close()`; any crash that happens in the recovery steps leaves the allocator dirty.

4.5.1 Filter Garbage Collection. Precise GC, of course, is impossible in C and C++, due to the absence of type safety. Conservative collection admits the possibility that some 64 b value will erroneously appear to point to a garbage block, resulting in a block that appears to be in use, despite the fact that it was not allocated (or was freed) during pre-crash execution—in effect, a memory leak. Arguably worse, conservative collection is incompatible with pointer tagging and other nonstandard representations. *Filter functions* serve to address these limitations.

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

Figure 4. Example of a filter function for binary tree nodes.

Figure 3 shows the basic variables and functions related to filter GC. The basic principle is that, when `getRoot <>()` is called after `init()` but before `recover()`, its type (obtained via template instantiation) is recorded in the transient array `rootsFunc`, in the form of a lambda expression that calls the `visit<T>()` function. Then in `recover()`, `collect()` traces all reachable blocks by calling `visit<T>()` iteratively from persistent roots until no more new blocks are found. Each `visit<T>()` function marks its block as reachable and then calls `filter<T>()`, which is assumed to call `visit<U>()` for each pointer of type `U` in the block.

For each type `U` used for persistent data, the programmer is encouraged to provide a corresponding `filter<U>()`. Figure 4 presents an example filter for binary tree nodes. If no `filter<U>()` has been specialized, the default conservative filter, defined in Figure 3, is called instead.

While the implementation shown here utilizes C++ templates, filter functions are easily adapted to pure C by arranging for `visit()` and `getRoot()` to take a pointer to the appropriate filter function as an extra parameter, and for filter functions themselves to pass the appropriate function pointer in each of their calls to `visit()`.

Note that the function pointers used in GC are reestablished in each execution, avoiding any complications due to recompilation or address space layout randomization (ASLR) [44]. Mechanisms to tag persistent roots with persistent type information are a potential topic for future work.

4.5.2 Sharing Across Processes. The mechanisms described above suffice to manage a persistent heap that is used by one application at a time. While this application may be multithreaded, its threads all live and die together. If we wish to allow a heap to be shared by threads in *different* processes—either with mutual trust or via protected libraries [19]—we must address a pair of problems. While neither is addressed in our current implementation, solutions appear straightforward.

First, if a heap in a newly rebooted system may be mapped into more than one process concurrently, we need a mechanism to determine which of these processes is responsible for recovery. While several strategies are possible, perhaps the simplest assigns this task to a dedicated *manager process*. Such a process could be launched by any application that calls `init()` on a currently inactive segment.

Second, we must consider the possibility that a process may crash (due to a software bug or signal) while others

continue to use the heap. While nonblocking allocation ensures that the heap will remain usable and consistent, blocks may leak for the same reasons as in a full-system crash: they may be allocated but not yet attached, detached but not yet deallocated, held in a per-thread cache, or held in a limbo list awaiting safe reclamation. Given our reliance on post-crash garbage collection, these blocks can be recovered only by tracing from persistent roots. As indicated at the end of Section 3, we assume that crashes are rare and that it will be acceptable to implement blocking, "stop-the-world" collection when they occur. A likely implementation would employ a failure detector provided by the operating system and fielded by the manager process mentioned above. In the wake of a single-process crash, the manager could initiate stop-the-world collection using a quiescence mechanism adapted from asymmetric locking [49].

4.6 Position Independence

There are several reasons not to implement pointers as absolute virtual addresses in persistent memory. If an application uses more than one independent persistent data structure, the addresses of those structures will need to be distinct. If new applications can be designed to use arbitrary existing structures, then every such structure would need to have a globally unique address range, suggesting the need for global management and interfering with security strategies like ASLR.

One option, employed by our earlier work on InterWeave [7], is to explicitly relocate a heap when it is first mapped into memory, "swizzling" pointers as necessary. Unfortunately, this approach requires precise type information and still requires that all concurrent users map a heap at the same virtual address.

Some systems (e.g., PMDK [41]) use offsets from the beginning of a destination segment rather than absolute addresses, and specify the ⟨base, offset⟩ pair in 128 bits. This *based pointer* convention requires that the starting address of the segment be available (e.g., in a reserved register) in order to convert a persistent pointer to a virtual address. An attractive alternative, used by NV-Heaps [10] is to calculate the offset not from the beginning of the segment but from the location of the pointer itself, since that location is certain to be conveniently available when storing to or loading from it. Chen et al. [8] call such offset-based pointers *off-holders*.

We implement off-holders as 64-bit `pptr<T>` smart pointers in C++, and instruct programmers to use them instead `T*` pointers. All of the usual pointer operations work as one would expect, with no additional source-code changes. Chen et al. [8] report overheads for this technique of less than 10%.

For cross-region metadata pointers within the same instance of Ralloc (e.g., persistent roots that reside in the metadata region and point to the superblock region), `pptr` takes an optional template parameter as the index of a region. The default value indicates that this is an off-holder pointing to a target in the current region. Three other values can be used to indicate a based pointer for the metadata, descriptor, or superblock region of the segment. Ralloc records the base address of regions during initialization, allowing it to look up these addresses while converting a region-specific pointer to an absolute address. Note that based pointers are never needed by application programmers; they appear only within the code of Ralloc itself.

Given a hard limit on the size of a superblock region (currently 1 TB), Ralloc is able to repurpose some of the bits in a 64 b `pptr`. As noted in Section 4.2, part of each list head is used for an anti-ABA counter. For an off-holder, the unused bits hold an arbitrary uncommon pattern that is masked away during use; this convention serves to reduce the likelihood that frequently-occurring integer constants will be mistaken for off-holders during conservative post-crash GC.

The `pptr` implementation does not currently support general cross-heap references. Among our near-term plans is to implement a *Region ID in Value (RIV)* [8] variant of `pptr`, retaining the smart pointer interface and the size of 64 bits.

5 Correctness

During crash-free execution, LRMalloc is both safe and live: blocks that are concurrently in use (`malloc`ed and not yet `free`d) are always disjoint (no conflicts), and blocks that are `free`d are eventually available for reuse (no leaks). We argue that Ralloc preserves these properties, and is additionally lock free and recoverable.

Theorem 5.1 (Overlap freedom). *Ralloc does not overlap any in-use blocks.*

Proof sketch. This property is essentially inherited from LRMalloc. All small allocations are eventually fulfilled from thread-local caches, which are recharged by removing superblocks from the global free and partial lists. Large allocations, likewise, are fulfilled with entire superblocks, obtained using CAS to update the `used` size field. Only the CASing thread has the right to allocate from the new superblocks.

The global lists are lock-free Treiber stacks [46]. Only one thread at a time—the one that removes a superblock from a global list—can allocate from the superblock. Expansion of the heap (to create new superblocks) likewise happens in a single thread, using CAS to update the `used` size field.

Small blocks in separate superblocks are disjoint, as are blocks within a given superblock. The blocks that tile a superblock change size only when the superblock cycles through the free list; the superblocks that comprise a large allocation likewise cycle through the free list. Thus blocks of different sizes that overlap in space never overlap in time.

A thread that cannot allocate from a given superblock may still *deallocate* blocks, but the free list within the superblock again functions as a Treiber stack, with competing operations mediated by CASes on the anchor of the corresponding

descriptor. These observations imply that allocations of the same block never overlap in time. □

Theorem 5.2 (Leakage freedom). *Freed blocks in Ralloc eventually become available for reuse.*

Proof sketch. Reasoning here is straightforward. Small blocks, when deallocated, are returned to the thread-local cache. Superblocks are returned to global partial lists or (if `EMPTY`) to the global free list when the thread-local cache is too large, or when a large block is deallocated. In either case, deallocated blocks are available for reuse (typically by the same thread; sometimes by any thread) as soon as `free` returns. Exactly *when* reuse will occur, of course, depends on the pattern, across threads, of future calls to `malloc` and `free`. Note that safe memory reclamation [31, 51], if any, is layered *on top* of `free`: the Ralloc operation is invoked not at retirement, but at eventual reclamation. □

Theorem 5.3 (Recoverability). *Ralloc is recoverable.*

Recall that an allocator is *recoverable* if it ensures, in the wake of post-crash recovery, that the metadata of the allocator will indicate that all and only the "in use" blocks are allocated. For Ralloc, "in use" blocks are defined to be those that are reachable from a specified set of persistent roots. In support of this definition, Ralloc assumes that the application follows certain rules. Specifically:

1. It is (buffered) durably linearizable (Ralloc does not transform a transient application to be persistent).
2. It registers persistent roots in such a way that all blocks it will ever attach in the future will be reachable.
3. It eventually attaches every allocated block to the structure to make it reachable.
4. It eventually calls `free` for every detached block.
5. It specializes a filter function for any block whose internal pointers are not 64-bit aligned `pptrs`.

Proof sketch. These rules ensure that the application never leaks blocks during crash-free operation (rules 3 and 4), and that it enables GC-based recovery (rules 2 and 5). The size information of any in-use block in a descriptor is safely available to recovery via the `size class` and `block size` persistent fields. Assuming that pointers in types without specialized filter functions are always aligned and in `pptr` format, Ralloc's garbage collection, with or without specialized filter functions, is guaranteed to find all in-use blocks by tracing from the persistent roots. Having identified these blocks, Ralloc re-initializes its metadata accordingly, updating each descriptor and reconstructing free lists and partial lists. By the end of the recovery, all and only the in-use blocks are allocated (where "in use" is defined to include all blocks found by the collector, even if they were not actually `malloced` during pre-crash execution). □

Theorem 5.4 (Liveness). *Ralloc is lock free during crash-free execution.*

Proof sketch. The only unbounded loops in Ralloc occur in the Treiber-stack–like operations on the superblock free and partial lists and the free lists of individual superblocks, and in operations on anchors and the `used` size field. In all cases, the failure of a CAS that triggers a repeat of a loop always indicates that another thread has made forward progress.

Significantly, there are no explicit system calls (e.g., to `mmap`) inside Ralloc's allocation and deallocation routines. We assume that implicit OS operations, such as those related to demand paging and scheduling, always return within a reasonable time; we do not consider them as sources of blocking in Ralloc. □

6 Experiments

6.1 Setup

We ran all tests on Linux 5.3.7 (Fedora 30), on a machine with two Intel Xeon Gold 6230 processors, each with 20 physical cores and 40 hyperthreads. Threads were first pinned one per core in the first socket, then on the extra hyperthreads, and finally on the second socket. All experiments were conducted on 6 channels of 128 GB Optane DIMMs, all in the first socket's NUMA domain. Persistent allocators ran on an EXT4-DAX filesystem built on NVM; transient allocators ran directly on raw NVM [25].

We compared Ralloc to representative persistent and transient allocators including *Makalu* [20], *libpmemobj-1.6* from *PMDK* [41], *JEMalloc* [15], and *LRMalloc* [28] (Ralloc without flush and fence). Since all benchmarks and applications in our experiments used `malloc`/`free`, for PMDK's `malloc-to`/`free-from` interface we had to create a local dummy variable to hold the pointer for easy integration. For all tests we report the average of three trials. The source code for Ralloc is available at https://github.com/qtcwt/ralloc.

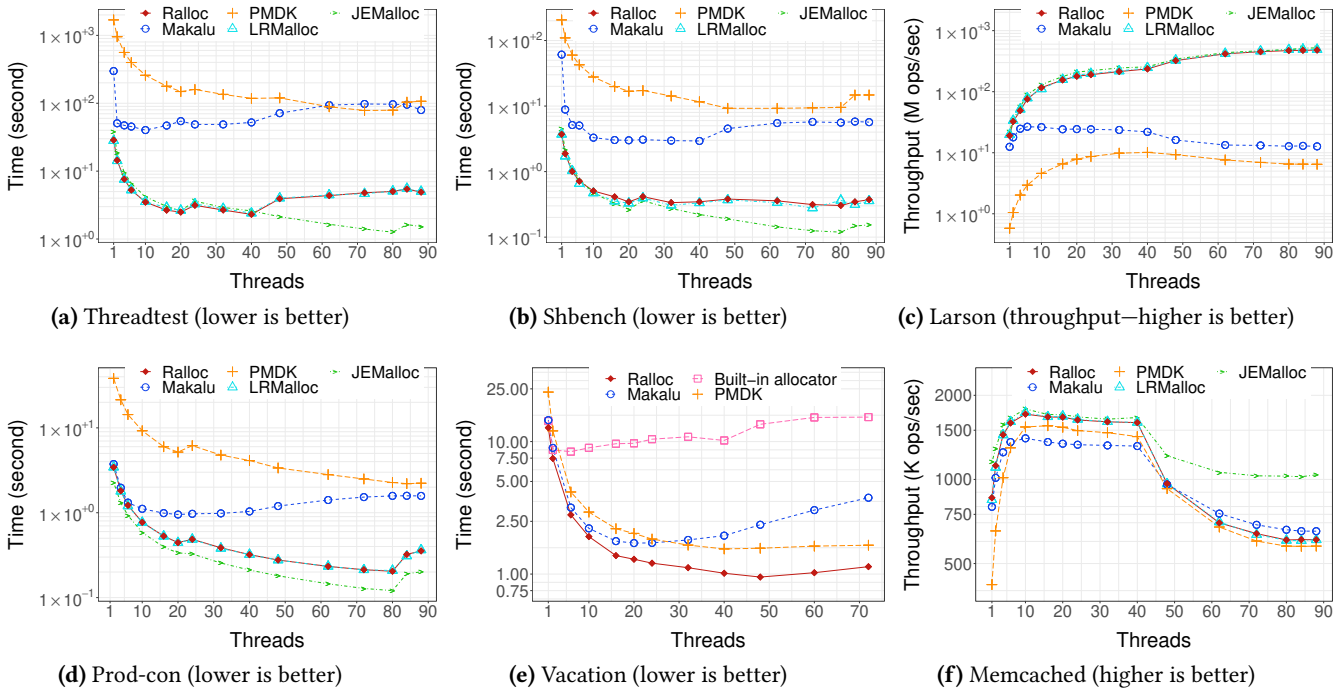
6.2 Benchmarks

Our initial evaluation employed four well known allocator workloads.

Threadtest, introduced with the *Hoard* allocator [2], allocates and deallocates a large number of objects without any sharing or synchronization between threads. In every iteration of the test, each thread allocates and deallocates 10^5 64-byte objects; our experiment comprises 10^4 iterations.

Shbench [34] is designed as an allocator stress test. Threads allocate and deallocate many objects, of sizes that vary from 64 to 400 bytes (the largest of Makalu's "small" allocation sizes), with smaller objects being allocated more frequently. Our experiment comprises 10^5 iterations.

Larson [26] simulates a behavior called "bleeding" in which some of the objects allocated by one thread are left to be freed by another thread. This test spawns t threads that randomly allocate and deallocate 10^3 objects in each iteration, ranging in size from 64 to 400 bytes. After 10^4 iterations, each thread creates a new thread that starts with



(a) Threadtest (lower is better) **(b)** Shbench (lower is better) **(c)** Larson (throughput—higher is better)

(d) Prod-con (lower is better) **(e)** Vacation (lower is better) **(f)** Memcached (higher is better)

Figure 5. Performance (log2 scaled). Each socket has a total of 20 two-way hyperthreaded cores.

the leftover objects and repeats the same procedure. Our experiment runs this pattern for 30 seconds.

Prod-con is a local re-implementation of a test originally devised for Makalu [3]. It is meant to assess performance under a producer-consumer workload. The test spawns $t/2$ pairs of threads and assigns a lock-free M&S queue [33] to each pair. One thread in each pair allocates $10^7 \cdot 2/t$ 64 B objects and pushes pointers to them into the queue; the other thread concurrently pops pointers from the queue and deallocates the objects.

Performance results appear in Figures 5a–5d. In many cases, curves change shape (generally for the worse) at 20 and 40 threads, as execution moves onto sister hyperthreads and the second socket, respectively. The 20-thread inflection point presumably reflects competition for cache and pipeline resources, the 40-thread inflection point the cost of cross-socket communication. Overall, Ralloc outperforms and scales better than PMDK and Makalu on all benchmarks, and is close to JEMalloc for low thread counts. Makalu, however, usually stops scaling before 20 threads.

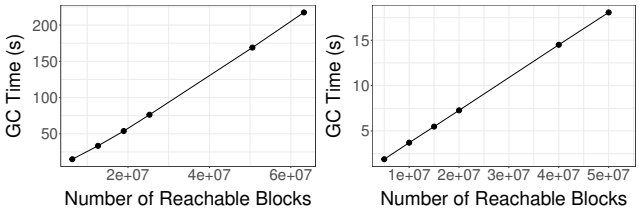
On Threadtest and Shbench, Ralloc performs around 10× faster than Makalu and PMDK, presumably because the earlier systems must log and flush multiple words in synchronized allocator operations, while Ralloc needs no logging at all, and flushes only occasionally—and then only a single word (the block size during superblock allocation).

On Larson, Ralloc performs up to 37× faster than Makalu. We attribute this to Makalu's lack of robustness for large numbers of threads. We have also (results not shown) tested the allocators on Larson with a wider range of sizes (64–2048 bytes, the largest "medium" allocation size in Makalu). In this test Makalu stopped scaling after only 2 threads, and performed up to 100× slower than Ralloc (1 M versus 100 M at 16 threads). This may suggest that "medium" allocations severely compromise Makalu's scalability.

On Prod-con, Ralloc's performance is close to that of Makalu for low thread counts, but afterwards scales better. This is because most of the time for low thread counts is spent synchronizing on the M&S queues, hiding the difference in allocation overhead.

6.3 Application Tests

We also tested Ralloc on a persistent version of *Vacation* and a local version of *Memcached*. Vacation (from the STAMP suite [5]) is a simulated online transaction processing system, whose internal "database" is implemented as a set of red-black trees. We obtained the code for this experiment, along with that of the Mnemosyne [50] persistent transaction system, from the University of Wisconsin's WHISPER suite [35]. Memcached [29] is a widely used in-memory key-value store. We modified it to function as a library rather than a stand-alone server: instead of sending requests over a socket, the client application makes direct function calls into the key-value code, much as it would in a library database like Silo [47]. Our version of memcached can also be shared safely between applications using the Hodor protected library system [19]; to focus our attention on allocator performance, we chose not to enable protection on library calls for the experiments reported here.



(a) Treiber Stack **(b)** Natarajan & Mittal Tree

Figure 6. GC Time Consumption.

The **Vacation** test employs a total of 16384 "relations" in its red-black trees. Each transaction comprises 5 queries, and the 10^6 transactions performed by each test target 90% of the relations. All queries are to create new reservations. Given that the code had been modified explicitly for persistence, we tested only the persistent allocators, which exclude LRMalloc and JEMalloc, but include Mnemosyne's built-in allocator, a persistent hybrid of Hoard [2] and DLMalloc [27].

The **Memcached** test runs the Yahoo! Cloud Serving Benchmark (YCSB) [13], configured to be write dominant (workload A [12] with 50% reads and 50% updates). In total, 5 M operations were executed on 1 M records.

Application performance results appear in Figures 5e and 5f. Results on Vacation resemble those of the allocator benchmarks: Ralloc scales better than other persistent allocators, and performs fastest for all sampled thread counts. Memcached tells a slightly more interesting story: Performance is relatively flat up to 40 threads, with Ralloc outperforming both Makalu and PMDK. Performance deteriorates with the cost of cross-socket communication, however, and Makalu gains a performance edge, outperforming Ralloc by up to 7% at 62 threads. Our best explanation is that Makalu provides slightly better locality for applications with a large memory footprint. In particular, instead of transferring an over-full thread-local free list (cache) back to a central pool in its entirety, as Ralloc does, Makalu returns only half, allowing the local thread to continue to allocate from the portion that remains.

On memcached's read-dominant workload (workload B [12] with 95% reads and 5% updates—not shown here), Ralloc continues to outperform Makalu by a small amount at all thread counts. The curves are otherwise similar to those in Figure 5f.

6.4 Recovery

In a final series of experiments, we measured the cost of Ralloc's recovery procedure by running an application without calling `close()` at the end, thereby triggering recovery at the start of a subsequent run.

We inserted random key-value pairs into a lock-free Treiber stack [46] in one experiment and, in the other, into the nonblocking binary search tree of Natarajan and Mittal [36]. The size of nodes in both is 64 B. We recorded recovery time for varying numbers of reachable blocks; results appear in Figure 6. As expected, recovery time is linear in the number of reachable blocks. Per-node time is higher in the case of the tree, presumably due to poorer cache locality. After recovery, the application was able to restore the structure correctly in all cases, and to continue performing operations without error.

We were unable to run Makalu's recovery routine in our local environment. Our recovery experiment, however, is similar to the one reported by Makalu's creators [3]. According to their paper, Makalu takes around 1 second, parallelized on 6 threads across persistent roots, to recover a collection of Treiber stacks when the longest chain is 50 K nodes and the data left in the heap is 1563 MB. Ralloc, on the other hand, takes about 17.5 seconds, single-threaded, to recover one Treiber stack of 50 M nodes, with 3052 MB left in the heap. These data suggest that Ralloc does not incur dramatically higher recovery cost than Makalu.

While we currently run recovery on a single thread, it would be straightforward in the procedure of Section 4.5 to parallelize Step 5 across persistent roots and Steps 6–9 across superblocks; we leave this to future work.

7 Conclusions

In this paper, we introduced the notion of *recoverability* as a correctness criterion for persistent memory allocators. Building on the (transient) LRMalloc nonblocking allocator, we then presented Ralloc, which we believe to be the first recoverable lock-free allocator for persistent memory.

As part of Ralloc, we introduced the notion of *filter functions*, which allow a programmer to refine the behavior of conservative garbage collection without relying on compiler support or per-block prefixing [11]. We believe that filter functions may be a useful mechanism in other (e.g., transient) conservative garbage collectors.

Using recovery-time garbage collection, Ralloc is able to achieve recoverability with almost no run-time overhead during crash-free execution. By using offset-based pointers, Ralloc supports *position-independent data* for flexible sharing across executions and among concurrent processes.

Experimental results show that Ralloc matches or exceeds the performance of Makalu, the state-of-the-art lock-based persistent allocator, and is competitive with the well-known JEMalloc transient allocator. Near-term plans include the addition of general cross-heap persistent pointers, integration with persistent libraries [19], parallelized and optimized recovery, and detection and (stop-the-world) recovery for independent process failures. Longer term, we plan to explore online recovery.

Acknowledgments

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

References

[1] H. A. Beadle, W. Cai, H. Wen, and M. L. Scott. Towards efficient nonblocking persistent software transactional memory. Technical Report TR 1006, Department of Computer Science, Univ. of Rochester, Apr. 2019. Extended abstract published as a brief announcement at *PPoPP* 2020.

[2] E. D. Berger, K. S. McKinley, R. D. Blumofe, and P. R. Wilson. Hoard: A scalable memory allocator for multithreaded applications. In *9th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 117–128, Cambridge, MA, Nov. 2000.

[3] K. Bhandari, D. R. Chakrabarti, and H.-J. Boehm. Makalu: Fast recoverable allocation of non-volatile memory. In *ACM SIGPLAN Intl. Conf. on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 677–694, Amsterdam, Netherlands, Oct. 2016.

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

[5] C. Cao Minh, J. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford transactional applications for multi-processing. In *IEEE Intl. Symp. on Workload Characterization (IISWC)*, pages 35–46, Seattle, WA, Sept. 2008.

[6] D. R. Chakrabarti, H.-J. Boehm, and K. Bhandari. Atlas: Leveraging locks for non-volatile memory consistency. In *ACM Intl. Conf. on Object Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 433–452, Portland, OR, Oct. 2014.

[7] D. Chen, C. Tang, X. Chen, S. Dwarkadas, and M. L. Scott. Multi-level shared state for distributed systems. In *Intl. Conf. on Parallel Processing (ICPP)*, pages 131–140, Vancouver, BC, Canada, Aug. 2002.

[8] G. Chen, L. Zhang, R. Budhiraja, X. Shen, and Y. Wu. Efficient support of position independence on non-volatile memory. In *50th IEEE/ACM Intl. Symp. on Microarchitecture (MICRO)*, pages 191–203, Cambridge, MA, Oct. 2017.

[9] D. Chinner. xfs: DAX support, Mar. 2015. https://lwn.net/Articles/635514/.

[10] J. Coburn, A. M. Caulfield, A. Akel, L. M. Grupp, R. K. Gupta, R. Jhala, and S. Swanson. NV-Heaps: Making persistent objects fast and safe with next-generation, non-volatile memories. In *16th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 105–118, Newport Beach, CA, Mar. 2011.

[11] N. Cohen, D. T. Aksun, and J. R. Larus. Object-oriented recovery for non-volatile memory. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):153:1–153:22, Oct. 2018.

[12] B. Cooper. YCSB core workloads, 2010. https://github.com/brianfrankcooper/YCSB/wiki/Core-Workloads.

[13] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *1st ACM Symp. on Cloud Computing (SoCC)*, pages 143–154, Indianapolis, IN, June 2010.

[14] A. Correia, P. Felber, and P. Ramalhete. Romulus: Efficient algorithms for persistent transactional memory. In *30th on Symp. on Parallelism in Algorithms and Architectures (SPAA)*, pages 271–282, Vienna, Austria, July 2018.

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

[16] K. Fraser. *Practical Lock-Freedom.* PhD thesis, King's College, Univ. of Cambridge, Sept. 2003. Published as Univ. of Cambridge Computer Lab technical report #579, Feb. 2004. https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-579.pdf.

[17] M. Friedman, M. Herlihy, V. Marathe, and E. Petrank. A persistent lock-free queue for non-volatile memory. In *23rd ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPoPP)*, pages 28–40, Vienna, Austria, Feb. 2018.

[18] A. Gidenstam, M. Papatriantafilou, and P. Tsigas. NBmalloc: Allocating memory in a lock-free manner. *Algorithmica*, 58(2):304–338, Oct 2010.

[19] 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 *USENIX Annual Technical Conf. (ATC)*, pages 489–504, Renton, WA, July 2019.

[20] Hewlett Packard Enterprise. makalu_alloc, May 2017. https://github.com/HewlettPackard/Atlas/tree/makalu/makalu_alloc.

[21] R. L. Hudson, B. Saha, A.-R. Adl-Tabatabai, and B. Hertzberg. McRT-Malloc—A scalable transactional memory allocator. In *Intl. Symp. on Memory Management (ISMM)*, pages 74–83, Ottawa, ON, Canada, June 2006.

[22] Intel Corp. *Intel 64 and IA-32 Architectures Software Developer's Manual*, May 2019. 325462-070US.

[23] J. Izraelevitz, T. Kelly, and A. Kolli. Failure-atomic persistent memory updates via JUSTDO logging. In *21st Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 427–442, Atlanta, GA, Apr. 2016.

[24] J. Izraelevitz, H. Mendes, and M. L. Scott. Linearizability of persistent memory objects under a full-system-crash failure model. In *Intl. Symp. on Distributed Computing (DISC)*, pages 313–327, Paris, France, Sept. 2016.

[25] J. Izraelevitz, J. Yang, L. Zhang, J. Kim, X. Liu, A. Memaripour, Y. J. Soh, Z. Wang, Y. Xu, S. R. Dulloor, J. Zhao, and S. Swanson. Basic performance measurements of the Intel Optane DC persistent memory module. *CoRR*, abs/1903.05714, 2019.

[26] P. Larson and M. Krishnan. Memory allocation for long-running server applications. In *1st Intl. Symp. on Memory Management (ISMM)*, pages 176–185, Vancouver, BC, Canada, Oct. 1998.

[27] D. Lea. A memory allocator, Apr. 2000. http://gee.cs.oswego.edu/dl/html/malloc.html.

[28] R. Leite and R. Rocha. LRMalloc: A modern and competitive lock-free dynamic memory allocator. In *13th Intl. Meeting on High Performance Computing for Computational Science (VECPAR)*, pages 230–243, São Pedro, São Paulo, Brazil, Sept. 2018.

[29] libMemcached.org. libmemcached, 2011. https://libmemcached.org/libMemcached.html.

[30] Q. Liu, J. Izraelevitz, S. K. Lee, M. L. Scott, S. H. Noh, and C. Jung. iDO: Compiler-directed failure atomicity for nonvolatile memory. In *51st IEEE/ACM Intl. Symp. on Microarchitecture (MICRO)*, pages 258–270, Fukuoka, Japan, Oct. 2018.

[31] M. M. Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Transactions on Parallel and Distributed Systems*, 15(6):491–504, June 2004.

[32] M. M. Michael. Scalable lock-free dynamic memory allocation. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, pages 35–46, Washington DC, June 2004.

[33] M. M. Michael and M. L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *15th ACM Symp. on Principles of Distributed Computing (PODC)*, pages 267–275, Philadelphia, PA, May 1996.

[34] MicroQuill, Inc. shbench, 2014. http://www.microquill.com/smartheap/shbench/.

[35] S. Nalli, S. Haria, M. D. Hill, M. M. Swift, H. Volos, and K. Keeton. An analysis of persistent memory use with WHISPER. In *22nd Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 135–148, Xi'an, China, 2017. ACM.

[36] A. Natarajan and N. Mittal. Fast concurrent lock-free binary search trees. In *19th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPoPP)*, pages 317–328, Orlando, FL, Feb. 2014.

[37] F. Nawab, J. Izraelevitz, T. Kelly, C. B. Morrey III, D. R. Chakrabarti, and M. L. Scott. Dalí: A periodically persistent hash map. In *Intl. Symp. on Distributed Computing (DISC)*, volume 91, pages 37:1–37:16, Vienna, Austria, Oct. 2017.

[38] I. Oukid, D. Booss, A. Lespinasse, W. Lehner, T. Willhalm, and G. Gomes. Memory management techniques for large-scale persistent-main-

memory systems. *Proceedings of the VLDB Endowment*, 10(11):1166–1177, Aug. 2017.

[39] P. Ramalhete, A. Correia, P. Felber, and N. Cohen. OneFile: A wait-free persistent transactional memory. In *49th IEEE/IFIP Intl. Conf. on Dependable Systems and Networks (DSN)*, pages 151–163, Portland, OR, June 2019.

[40] A. Rudoff. Persistent memory programming. *Login: The Usenix Magazine*, 42:34–40, 2017.

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

[42] M. L. Scott. *Shared-Memory Synchronization*. Morgan & Claypool, 2013.

[43] S. Seo, J. Kim, and J. Lee. SFMalloc: A lock-free and mostly synchronization-free dynamic memory allocator for manycores. In *2011 Intl. Conf. on Parallel Architectures and Compilation Techniques (PACT)*, pages 253–263, Galveston, TX, Oct. 2011.

[44] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh. On the effectiveness of address-space randomization. In *11th ACM Conf. on Computer and Communications Security (CCS)*, pages 298–307, Washington DC, Oct. 2004.

[45] The Open Group. `pthread_mutex_lock`. IEEE Std 1003.1-2017, 2018.

[46] R. K. Treiber. Systems programming: Coping with parallelism. Technical Report RJ 5118, IBM Almaden Research Center, Apr. 1986.

[47] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden. Speedy Transactions in Multicore In-memory Databases. In *24th ACM Symp. on Operating Systems Principles (SOSP)*, pages 18–32, Farmington, PA, Nov. 2013.

[48] A. Vahldiek-Oberwagner, E. Elnikety, N. O. Duarte, M. Sammler, P. Druschel, and D. Garg. ERIM: Secure, efficient in-process isolation with protection keys (MPK). In *28th USENIX Security Symp. (SEC)*, pages 1221–1238, Santa Clara, CA, Aug. 2019.

[49] N. Vasudevan, K. S. Namjoshi, and S. A. Edwards. Simple and fast biased locks. In *19th Intl. Conf. on Parallel Architectures and Compilation Techniques (PACT)*, pages 65–74, Vienna, Austria, Sept. 2010.

[50] H. Volos, A. J. Tack, and M. M. Swift. Mnemosyne: Lightweight persistent memory. In *16th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 91–104, Newport Beach, CA, Mar. 2011.

[51] H. Wen, J. Izraelevitz, W. Cai, H. A. Beadle, and M. L. Scott. Interval-based memory reclamation. In *23th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPoPP)*, pages 1–13, Vienna, Austria, 2018.

[52] M. Wilcox. Add support for NV-DIMMs to Ext4, Dec. 2017. https://kernelnewbies.org/Ext4.

[53] D. Williams. Persistent memory, Aug. 2019. https://nvdimm.wiki.kernel.org/.

[54] J. Xu and S. Swanson. NOVA: A log-structured file system for hybrid volatile/non-volatile main memories. In *14th USENIX Conf. on File and Storage Technologies (FAST)*, pages 323–338, Santa Clara, CA, Feb. 2016.

[55] J. Yang, J. Izraelevitz, and S. Swanson. Orion: A distributed file system for non-volatile main memory and RDMA-capable networks. In *17th USENIX Conf. on File and Storage Technologies (FAST)*, pages 221–234, Boston, MA, Feb. 2019.