Makalu: Fast Recoverable Allocation of Non-volatile Memory

Kumud Bhandari *
Rice University and Hewlett Packard Labs, USA
kumudb@rice.edu

Dhruva R. Chakrabarti
Hewlett Packard Labs, USA
dhruva.chakrabarti@hpe.com

Hans-J. Boehm
Google Inc, USA
hboehm@google.com

Abstract

Byte addressable non-volatile memory (NVRAM) is likely to supplement, and perhaps eventually replace, DRAM. Applications can then persist data structures directly in memory instead of serializing them and storing them onto a durable block device. However, failures during execution can leave data structures in NVRAM unreachable or corrupt. In this paper, we present Makalu, a system that addresses non-volatile memory management. Makalu offers an integrated allocator and recovery-time garbage collector that maintains internal consistency, avoids NVRAM memory leaks, and is efficient, all in the face of failures.

We show that a careful allocator design can support a less restrictive and a much more familiar programming model than existing persistent memory allocators. Our allocator significantly reduces the per allocation persistence overhead by lazily persisting non-essential metadata and by employing a post-failure recovery-time garbage collector. Experimental results show that the resulting online speed and scalability of our allocator are comparable to well-known transient allocators, and significantly better than state-of-the-art persistent allocators.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors; D.4.2 [*Operating Systems*]: Storage Management; D.4.5 [*Operating Systems*]: Reliability

General Terms Languages, Performance, Reliability

Keywords non-volatile memory, persistent memory management, allocation, deallocation, garbage collection

* Work done at Hewlett Packard Labs. This author was partially supported by the US Department of Energy under Cooperative Agreement no. DE-SC0012199.

OOPSLA'16, November 2–4, 2016, Amsterdam, Netherlands
© 2016 ACM. 978-1-4503-4444-9/16/11...$15.00
http://dx.doi.org/10.1145/2983990.2984019

1. Introduction

Limitations in current DRAM technology scaling [4, 30] have prompted research in alternate memory technologies. Almost all alternatives being explored such as Memristor [38], Phase Change Memory (PCM) [29, 34], and 3D XPoint [33] are non-volatile in nature. Such non-volatile memory (NVRAM) combines the byte-addressability of DRAM with the persistence of hard disks. In the near future, NVRAM may at least partially replace DRAM, making persistent data accessible through CPU load and store instructions. With NVRAM, persistent data can be manipulated in the same format as stored, thus requiring no expensive and cumbersome conversion.

NVRAM opens up an opportunity to have in-memory object persistence so that program states that outlive the creating process can be preserved, shared, and reused [14, 15, 39]. Using this persist-and-reuse model, a quick restart of an application from an intermediate state appears a reality. While starting, an application looks for existing data that it can reuse. If present, the application adjusts its context and instead of computing from scratch, merely reuses the existing data for the rest of its computation.

Recently, a lot of work has gone into failure-resilience of persistent data [2, 3, 5, 14, 15, 39]. Persistent data must be updated with great care so as to be reusable. Otherwise updates may become visible to NVRAM only partially, or not in the intended order. This may happen as a result of failures in the middle of critical sections, volatile CPU caches, or out-of-order cache evictions. As a result, an interruption such as a power failure may introduce inconsistencies (e.g. dangling pointers) to persistent data in NVRAM. Hence, some set of critical updates to persistent data must be applied on all-or-nothing basis, and in the correct order with respect to other updates, to guarantee the consistency of in-memory persistent data. Prior work [14, 15, 39] in this area has focused on developing non-volatile memory programming libraries (NVMPLs) which enable programmers to specify a failure-atomic section (FASE) [14] of updates to persistent data using familiar programming languages such as C/C++.

This paper focuses on efficient memory management of persistent memory, an issue that was largely sidestepped by previous work. In addition to ensuring that online allocations

and deallocations are efficient, two primary challenges are addressed here:

1. Allocator metadata consistency: The internal invariants have to be maintained in NVRAM in a fail-safe manner across restarts and tolerated failures to continue to allocate memory correctly.
2. Failure-induced persistent memory leaks: If persistent memory is allocated but a failure occurs before an application handle can be assigned to that memory, that memory location is essentially leaked since it is unreachable on program restart.

This paper describes Makalu[1], a persistent memory allocator that uses offline garbage collection (GC) to provide leak-freedom. Persistent data has two distinct phases with respect to a mutator application: (1) ***online***, when the mutator is active and (2) ***offline***, when the mutator is absent, but the heap is still accessible. After a failure, once the NVMPL restores persistent data to a consistent state, Makalu performs the parallel mark phase of mark-and-sweep GC offline followed by incremental sweeps online to avoid both failure-induced and programmer-induced leaks. This approach enables us to interoperate with other NVMPLs, while supporting unrestricted online usage of standard de-/allocation interfaces within a C/C++ program.

Makalu has built-in failure-resilience and recovery mechanisms to guarantee consistency of its metadata without depending on any NVMPL. Note that Makalu does not determine what data needs to be persisted or the structure of the persistent data. It only determines the structure of the heap in NVRAM.

Our approach achieves interoperability with existing code while providing a safety-net against all sources of memory leaks. By having the offline GC restore certain metadata, we avoid the need to ensure full consistency of metadata at every allocation, normally an expensive online operation. Makalu's raw online allocation speed and multi-threaded scalability are comparable to well known transient allocators, a tremendous improvement compared to state-of-the-art persistent allocators. We integrated it with two NVMPLs, Atlas and Mnemosyne, and saw up to 2x speed improvement in some scientific applications compared to NVMPLs' default allocators. This observation leads us to believe that offline garbage collection may become an integral part of future NVRAM memory management schemes. Our contributions include:

- A careful analysis of the requirements and opportunities for NVRAM memory allocation.
- A set of techniques that result in an NVRAM memory allocator that is often competitive with widely used DRAM allocators, and provides failure consistency at orders of magnitude lower cost than prior approaches.
- A demonstration that recovery time garbage collection not only simplifies the programming model, but also, by allowing reconstruction of metadata at recovery time, greatly speeds up NVRAM memory allocation. The offline GC is performed at a small fraction of the cost of general GC.

[1] Makalu is the fifth highest mountain in the world. Its four-sided pyramid shape represents the challenges we simultaneously address: memory management, garbage collection, persistent vs transient data, and failure-resilience.

2. Background

2.1 Architectural Assumptions

We make similar assumptions about the underlying architecture as found in prior NVMPL work [14, 15, 39]. NVRAM is assumed to retain data through tolerated failures, such as power failures. For convenience, NVRAM latency is assumed to be comparable to DRAM, which may or may not be present. At the lowest level, initial NVRAM devices may make trade-offs resulting in write latencies appreciably longer than DRAM. But this increased latency may not be user-visible. It appears likely that memory controllers will have enough capacitive power backup such that write requests, once accepted by the memory controller, can be viewed as persistent. Even if the power and the CPU die, the controller will ensure that accepted requests are written. Hence, the actual write latency to the NVRAM device may not matter for our purposes.

NVRAM endurance is expected to be orders of magnitude better than that of SSDs [4, 33] but solutions have been proposed [19] if wear-out is a concern. We do not address NVRAM endurance in this paper.

Several levels of volatile caches and potentially other volatile buffers exist between the CPU and NVRAM. We assume a tolerated failure (such as a process crash, an OS kernel panic, and a power failure) to be fail-stop. In such an event, the data that are already in NVRAM survive, but other data in volatile hardware structures do not.

Instructions are available to selectively evict or flush a cache line from the volatile caches into NVRAM (such as Intel x86 CLFLUSH [26] and CLWB [25]). Once evicted, these cache lines will eventually reach memory. These instructions are expensive [14, 39] and should be sparingly used[2].

2.2 Programming Assumptions

NVRAM is mapped directly into the process address space without any buffering and is accessible using CPU loads and stores. Persistent data is stored in named containers (a concept similar to mmapped files) called NVRAM regions. A persistent heap exists within the realm of an NVRAM region. When a heap is in a consistent state offline, all useful data that should be accessed upon restart must be reachable from a set of known persistent roots, otherwise, data can be considered garbage and reclaimed. In Makalu, there are 512

[2] A CLFLUSH instruction [26] in Intel x86-64 takes $\approx 200ns$ [14].

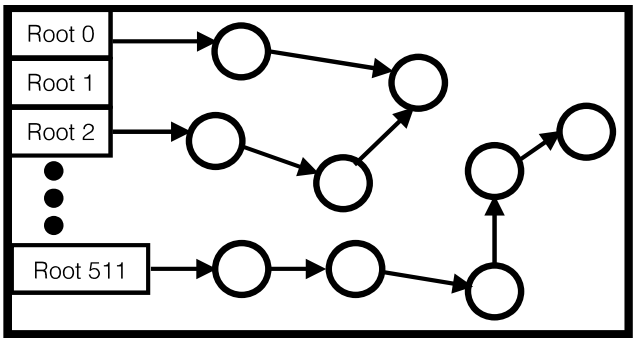


Figure 1: A snapshot of persistent heap with heap objects reachable from top level region roots within a NVRAM region.

top-level region roots stored in known locations within the NVRAM region. These assumptions are similar to ones made by most current NVMPLs [2, 14, 15, 39]. Figure 1 shows a sample persistent heap along with top level roots. We assume that an NVRAM region is always mapped at the same base address so that existing persistent-to-persistent pointers embedded in it remain valid, while persistent-to-transient pointers are ignored by the offline GC. With more precise pointer identification in the future, the offline GC phase can automatically clear such pointers for the programmer.

We assume that Makalu will be used in conjunction with other NVMPLs. The usage model requires the programmer to identify persistent data and manage them within an NVRAM region using malloc/free-like calls. Like much recent research in this area [14, 39], we assume an explicit persistence model where persistent data are directly identified by the programmer at allocation time. In our framework, this is done by using our APIs to allocate such data. This is in contrast to reachability-based persistent schemes [7, 16, 23] where programmers are not required to indicate persistence at object allocation time; instead, all data reachable from a persistent root must be made persistent. Failure-resilience of persistent data has a cost and the programmer may not want arbitrary data, such as passwords, to be transparently persisted. Thus we choose to favor explicit programmer control, also avoiding implicit reachability scans during execution.

Note that Makalu only guarantees the consistency of persistent heap structure, and not of the data stored within. The latter is expected to be provided by the NVMPL in use.

2.3 Terminology

Some commonly used memory allocation terms have the following meaning in this paper:

Memory object: A contiguous sequence of bytes in a persistent heap. The starting address is returned by the allocator while fulfilling a memory allocation request and the size corresponds to the number of bytes actually allocated.

Granule: The unit of actual memory allocated. In Makalu, all memory requests are rounded up to some multiple of the granule size, which is 16-bytes.

Block: A larger fixed-size contiguous sequence of bytes of virtual address space that can be divided into smaller memory objects to fulfill memory requests. In Makalu, the default size of a block is 4096 bytes (same as the page size in common operating systems). The starting address of a block is always page-aligned in Makalu.

Chunk: A contiguous sequence of one or more block(s).

3. NVRAM Allocator Challenges

Traditional factors, such as memory consumption and allocation speed, have historically influenced the design of transient memory allocators. Designing a persistent memory allocator for NVRAM offers the following additional challenges.

3.1 Failure-Induced Inconsistencies

A failure in the middle of updating the allocator's metadata can lead to metadata inconsistencies. We categorize such inconsistencies as ***internal*** w.r.t the allocator. Such internal inconsistencies often have disastrous consequences such as a failure to restart properly, erroneous re-allocation of memory objects currently in use, or the leakage of large chunks of memory.

A failure may also cause discrepancies at the mutator/allocator interface. We classify such inconsistencies as ***external*** w.r.t the allocator. For example, consider a scenario where a failure occurs after a call to malloc has returned (i.e. after an allocator has updated the internal metadata in a fail-safe manner) but before the returned address is stored in a persistent location by the mutator. This is essentially a failure-induced memory leak since upon restart, the allocator deems the returned memory object "allocated" though it is not reachable from any persistent root.

3.2 Transient vs. Persistent Metadata

We classify NVRAM allocators' metadata into two categories: ***core*** and ***auxiliary***. Core metadata is irrecoverable once corrupted. Auxiliary metadata on the other hand can be re-created using consistent core metadata. Although auxiliary metadata can be re-created at the beginning of each restart, and maintained in transient memory to avoid the cost of failure consistency, doing so may cause a long restart time, especially if recreating such auxiliary metadata is time-consuming. On the other hand, maintaining it in NVRAM may make restart instantaneous but at the expense of online overhead to maintain its failure consistency.

3.3 Online Failure Consistency Overhead vs. Recovery

Core persistent metadata often need to be updated in a way that always ensures their consistency. Such a mechanism frequently uses expensive instructions to flush cache lines after each update, and can adversely affect the allocation speed.

The consistency of auxiliary metadata stored in NVRAM need not be guaranteed as aggressively as the core metadata

because it can be recreated from core metadata. For instance, an allocator may choose to only periodically flush cache lines containing updates to auxiliary metadata rather than after every update, so long as any resulting inconsistency is detectable. This approach reduces the online consistency overhead but increases the recovery time. If a failure occurs when the metadata is momentarily inconsistent, it has to be re-computed in the recovery phase. Hence, tradeoffs exist between recovery time vs. the online consistency overhead.

3.4 Safe Deallocation

Many of the existing NVMPLs such as Mnemosyne[39] and Atlas[14] use transactional logging mechanism to guarantee failure-atomic updates within a FASE. When a persistent allocator is used in conjunction with such an NVMPL in a multi-threaded application, it has to handle deallocation requests from within a FASE in the following special manner: the deallocation of the object itself and its reuse to fulfill future memory requests have to be delayed until the allocator can confirm that the deleted memory object will never become live in the future. Logs belonging to partially executed FASEs could potentially hold the memory object's address. When the log is replayed during recovery to restore the consistency of user data, the memory object may be accessed via the reference in the log despite the program's request to deallocate it earlier during the online execution phase. Memory allocators for programs written using software transactional memory have to tackle a similar problem (see [24]). A persistent allocator needs to offer a mechanism to delay deallocation requests until the NVMPL in use can confirm that the deallocations can be done safely.

4. Overview of our Approach

4.1 Integrated De-/allocation and Garbage Collection

Makalu is built upon the Boehm-Demers-Weiser garbage collector (**bdwgc**) [12]. We chose to build our work upon the bdwgc framework instead of other transient allocators, such as Hoard [9] or jemalloc [18], because our approach relies heavily on garbage collection, and bdwgc provides inherent support for it. Grafting a garbage collector onto a separately designed allocator is nontrivial, and in fact, quite complex [35]. Having to deal with the persistence of both sets of data structures in Makalu's case makes such grafting even more complicated.

However, Makalu and bdwgc differ in several key aspects. In addition to the transformation needed for the allocator to become failure-resilient and function correctly across restarts, Makalu and bdwgc have some major differences in allocation and deallocation strategies. As bdwgc was designed for automatic memory management online, it has poor support for explicit deallocations. Explicit online deallocation is important in Makalu as we only support GC offline. While bdwgc supports thread-local allocations, it does not support thread local deallocation. Each deallocated object is returned to the global freelist which requires holding a global lock. This approach lacks multi-threaded scalability and prevents immediate memory reuse. Makalu, on the other hand, supports thread-local deallocations, and also uses a simple strategy to tackle the issue of memory blowup in multi-threaded applications, when some threads favor allocation while others favor deallocation (see § 7). This was not a concern in bdwgc in the absence of thread-local deallocations. We compare Makalu with the intermediate modified version of bdwgc in our evaluation (see § 13). Our results show that Makalu has the ability (rare among allocators) to support both explicit deallocation and garbage collection efficiently.

Similar to bdwgc, Makalu's heap is structured as a Big-Bag-of-Pages, maintaining persistent metadata only at the block level and in separate headers (see § 5.1). This approach helps minimize the amount of persistent metadata. Core information stored in the header about the layout of the heap (e.g. object size associated with a block) is considered to be irrecoverable once lost, and hence is only updated with ACID guarantees. We rely on locks for isolation and a built-in log-based approach for ensuring all-or-nothing visibility for a set of updates to NVRAM (§ 8).

In our setting, we currently run the garbage collector in a fully conservative mode, with no pointer location or type information communicated to the collector. This suffices to ensure metadata consistency. As discussed in section 9, other choices are also possible. The notion of offline garbage collection to mitigate failure-induced leaks, improve interoperability, programmability and online allocation performance explored in this paper can be applied in the context of strongly-typed languages such as Java and in the presence of precise garbage collection as well.

4.2 Choosing Persistent Metadata

To reduce the cost of persistent metadata updates, Makalu maintains a list of free objects in transient memory (see § 6) during the online phase. An allocation and a deallocation only require the corresponding memory object to be respectively removed from and added to the transient freelist. A failure may cause outstanding memory objects in Makalu's transient freelist to be lost momentarily. Makalu uses offline garbage collection to reclaim such objects and fix all other failure induced external inconsistencies based on the reachability of memory objects in the persistent heap. As a positive side effect, it also reclaims persistent memory leaked due to programming errors.

To avoid expensive computation at clean non-failure restarts, Makalu stores some selected auxiliary information, such as the header look-up table (see § 5.2), in persistent memory. We minimize the expense of persistent updates to these structures during the online phase, by assuming them to be inconsistent after a failure, and rebuilding them offline from core persistent metadata. Thus, online updates to these structures only need to be visible in NVRAM by the time

Interface	User	Description
MAK_start	N	One time call to set up and start Makalu in a new NVRAM region
MAK_restart	N	Restart Makalu from existing metadata
MAK_start_off	N	Restart Makalu offline
MAK_collect	N	Request GC offline
MAK_close	N	Signal Makalu to stop gracefully
MAK_set_free_cb	N	Set callback function for free (see § 10)
MAK_safe_free	N	Execute deferred free(s) (see § 10)
MAK_malloc MAK_calloc MAK_realloc MAK_free	P	Drop-in replacement for standard C/C++ de-/allocation methods
MAK_set_root MAK_get_root	P	Sets/gets top level NVRAM region roots

Table 1: List of Makalu's public interfaces.
User: N = NVMPL, P = Programmer

Makalu stops gracefully. This assumption enables us to potentially accumulate some number of updates to persistent structures before having to guarantee visibility. (see § 8.1).

4.3 APIs Provided by Makalu

Table 1 presents a list of Makalu's major functions in its public interface. Programmers are only responsible for invoking a handful of these functions. For integration with a transaction-based NVMPL, Makalu provides interface to defer deallocation requests until the NVMPL in use can confirm that the deallocated object is truly unreachable (see § 3.4). While the programmer-facing interface is largely self-explanatory, we defer the discussion of deferred deallocation and NVMPL-facing interface until § 10.

4.4 Comparison with Existing NVRAM Allocators

In both Mnemosyne [39] and Atlas [14], persistent memory allocations must be done within FASEs in order to guarantee the absence of failure-induced memory leaks. In addition to overheads incurred by failure-atomicity requirements of a FASE for every allocation, this clearly is a programming constraint. NV-Heaps [15] provides automatic garbage collection using reference counting but requires weak pointers to correctly deal with cyclic data structures. Some existing NVRAM allocators, such as nvm_malloc [36], require two method calls just to allocate a persistent object. Other NVMPLs, such as pmem.io [2], combine allocation, initialization and publication steps into a single allocation method call. These interfaces are far removed from traditional allocation interfaces.

Avoiding failure-induced memory leaks requires building a consensus between the allocator and the NVMPL after failure regarding what allocated memory objects are in use vs. free. There are several ways to obtain this consensus. One simple approach, but a burdensome one for programmers, is to require explicit code that traverses persistent data structures and reports them to the allocator after a failure, so that others can be deallocated. Another approach, one that seems to be taken by existing NVMPLs, is to tightly coordinate de-/allocation actions with the failure consistency semantics as described earlier. We present a superior approach that relies on offline garbage collection by tracing through application data structures to identify reachable NVRAM locations, essentially reaching consensus with the NVMPL in use. This way, familiar de-/allocation interfaces remain unchanged and can be called anywhere in the program, both within and outside FASEs.

Figure 2 compares three versions of the same code snippet written using a generic abstraction of a FASE [14] to add a node to the persistent queue in a fail-safe and thread-safe manner, while using a different flavor of a persistent allocator for each version. With the NVMPL's default allocator as shown in figure 2a, allocation and publication must happen within the same FASE to avoid failure-induced memory leaks, resulting in a large FASE. Another stand-alone persistent allocator, nvm_malloc [36] as shown in figure 2b supports reserving and initializing a persistent memory object before entering the FASE. However, the actual allocation and publication steps must still occur within the FASE as shown in figure 2b because nvm_malloc combines allocation and publication steps into a single allocation method. The primary inconvenience of this approach is the non-traditional interface. The use of Makalu, as shown in figure 2c, results in the smallest FASE, and a more familiar programming paradigm. In fact, while going from a transient to a persistent version of this code, the only change necessary in the Makalu-enabled version is the replacement of the allocation call with Makalu's corresponding one.

Note that the primary goal of our work is to understand the challenges of developing an interoperable and leak-free allocator, and investigate design decisions that can minimize failure consistency overhead. Improving the raw performance of an allocator is only a secondary goal.

5. Internal Structures and Layouts

In this section, we describe the internal structures of Makalu and how they are laid out in persistent memory within the NVRAM region. We classify each structure as either ***core*** or ***auxiliary*** metadata as described in § 3.2.

5.1 Persistent Block Header

Both an individual block as a well as a contiguous set of blocks (a chunk) in Makalu have a persistent header associated with them. Figure 3 shows the important header fields. The fields hb_block, and hb_sz store the starting address of a block or a chunk and its total size respectively. The field hb_flag indicates whether a block or a chunk is currently

```
BEGIN_FASE();
/* allocate */
Node* t_tmp =
 nvm_alloc(sizeof(Node));
/* initialize */
t_tmp->val = 10;
t_tmp->next = NULL;
/* publish */
p_queue->tail->next = t_tmp;
p_queue->tail = t_tmp;
END_FASE();
```

(a) An NVMPL's default allocator [14, 39]

```
/* reserve */
Node* t_tmp =
 nvm_reserve(sizeof(Node));
/* initialize */
t_tmp->val = 10;
t_tmp->next = NULL;
/* allocate + publish */
BEGIN_FASE();
p_queue->tail->next = t_tmp;
nvm_activate(t_tmp,
  &(p_queue->tail), t_tmp,
  NULL, NULL);
END_FASE();
```

(b) nvm_malloc [36]

```
/* allocate */
Node* t_tmp =
 MAK_alloc(sizeof(Node));
/* initialize */
t_tmp->val = 10;
t_tmp->next = NULL;
/* publish */
BEGIN_FASE()
p_queue->tail->next = t_tmp;
p_queue->tail = t_tmp;
END_FASE()
```

(c) Makalu

Figure 2: Code snippets using different flavors of persistent allocators and a generic NVMPL abstraction of a FASE. Prefixes 't_', and 'p_' denote transient and persistent program variables respectively. Each code snippet shows the common programming idiom of allocation, initialization, and publication of a persistent node of a queue.

```
header {
  void* hb_block;
  long hb_sz;
  int hb_flag;
  long hb_mark_bits[BITS_SZ];
  long hb_n_mark_bits;
  ...
}
```

Figure 3: Structure of the block header

in use or free. If a block is currently used to fulfill memory requests, hb_sz stores the size of memory objects allocated from that block. Note that all objects allocated from a single block are of the same size in Makalu. We regard the above three fields in each header as part of the core metadata and the rest are auxiliary. Makalu updates these fields using built-in support for ACID guarantees described in § 8.

Each object in a block has a corresponding bit in hb_mark_bits. If the bit is set, the object is either already allocated or some thread has already added to its freelist intending to allocate it. Alternatively, a thread cannot add to its freelist an object within a block whose mark bit is already set.

Field hb_n_mark_bits stores the count of mark bits currently set for convenience purpose. Although mark bits are stored in persistent memory, we regard them as auxiliary metadata because Makalu has the ability to recreate them offline based on the reachability of objects using GC.

A block header is allocated within ***header spaces***, which are one or more fixed-length sections of memory within an NVRAM region (and outside the heap) specifically designated for this purpose and shown in figure 4. Allocating headers only in header spaces enables Makalu to precisely know where all the core information is located so that it can be found during recovery to recreate auxiliary metadata.

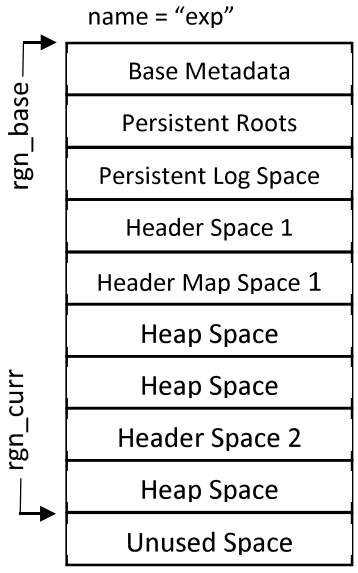


Figure 4: Layout of Makalu in an NVRAM region

Occasionally, one or more free adjacent blocks having their own headers coalesce to form a chunk requiring only a single header for the chunk beyond that point (see § 7.2). This block coalescing action frees up one or more headers which are added to the ***header freelist (HFL)*** for future reuse. HFL is auxiliary metadata as it can be recreated by scanning header spaces for all free headers.

5.2 Persistent Header Map

The header map in Makalu provides a method to conveniently look up corresponding header information for a given memory object, a block or a chunk. Additionally, it provides iterators to selectively iterate over blocks (free vs. allocated) via headers and without touching the actual blocks. This map

is adopted from bdwgc[11]. It is stored in ***header map space***, another specially designated section of memory within an NVRAM region for this purpose and shown in figure 4.

This map is auxiliary metadata that can be re-created in the header map space from scratch by adding each header from one or more header space(s). It is nonetheless stored in NVRAM to avoid the expense of rebuilding it at each normal re-start. A single scan through a header space during offline recovery is enough to create both the header map and the HFL. Therefore, we maintain their failure consistency less aggressively (see § 8.1) and rebuild them from scratch after each failure, which presumably should be rare.

5.3 Persistent Log Space

Makalu uses a log-based approach to update the core metadata in a fail-safe manner (see § 8). Since the persistent logs must be consistent at all times, the log space is a core data structure. As part of setting up a new region, it designates a fixed amount of space, as shown in figure 4, within the NVRAM region for persistent logs to be written. This space is reused repeatedly across execution cycles.

5.4 Persistent Roots

Makalu designates a ***persistent root space*** (separate from the heap space) within each NVRAM region for storing top-level NVRAM region roots. Makalu supports up to 512 top-level NVRAM region roots so that useful data within the NVRAM region's heap space can be conveniently accessed. Any i^{th} region root can be accessed using setter and getter methods listed in table 1.

5.5 Persistent Base Metadata

Apart from the information in the block header, Makalu also maintains the following information as part of the core metadata.

- NVRAM region base (`rgn_base`) and max heap (`rgn_curr`) address
- Log space starting address (`md_log_space_start`)
- Current log version (`md_log_version`) (see § 8)
- List of header spaces
- Address of the header map space
- Persistent root space start address

All of the above information is stored in base metadata space shown in figure 4. Most fields in base metadata, such as log space starting address, do not change once set. Other metadata fields, such as the current log version, are updated with ACID guarantees when necessary. Recovery is not possible without the consistent information provided by each field in the base metadata.

5.6 Transient Chunk Freelists

During the online phase, Makalu maintains a global transient list of free chunks segregated by the number of free blocks in them. When Makalu restarts, it recreates the list by adding all free chunks to it using the iterator provided by the header map.

5.7 Transient Object Freelists

Makalu classifies memory objects as *small* ($\leq$ 400 bytes), *medium* ($>$ 400 bytes, $\leq$ half a block), and *large* objects ($>$ half a block). During the online phase, Makalu maintains small object freelists on a per thread basis and global freelists for medium objects. Both small and medium object freelists are segregated by object size and each freelist is a transient LIFO list. Large object de-/allocations are handled directly by the chunk freelist (see § 6).

5.8 Transient Reclaim Lists

Makalu creates a list of partially allocated blocks (containing some free objects) at the beginning of each online phase by iterating over allocated blocks using the iterator provided by the header map. The reclaim list is a global transient structure organized as a set of object-size-segregated lists. Such a list enables us to sweep one block at a time looking for free objects of a particular size to refill a freelist.

6. Allocation

Allocation requests for small and medium objects are rounded up to the nearest granule multiple. For small objects, the allocating thread simply removes and returns the first item in the thread local freelist for that size. For medium sized objects, it removes and returns the first item from the appropriate global freelist after acquiring a lock for that size. Allocating objects from a transient freelist requires no persistent updates – note that the mark bit for the object is already set by this time (see below).

6.1 Refilling an Empty Freelist

If a freelist for a particular size is empty, the allocating thread performs an *incremental sweep*, i.e. it scans a partially allocated block to refill the freelist. It acquires a lock for a reclaim list corresponding to that size, removes the first block in the list and looks up the header for the block using the header map. Each free object, as indicated by its mark bit in the header, is added to the transient freelist while simultaneously setting the mark bit. A set mark bit indicates to another thread (going through the block for free objects at later times) that the object is either already in someone else's freelist or allocated.

A gracefully terminating thread gives each remaining object in its transient freelists back to the block by clearing the corresponding mark bit in the header for that object (so that other threads can use it to refill their freelist). During this process, if Makalu notices that a handful of objects have be-

come available in a specific block, it adds that block back to the reclaim list making it available to other threads for sweeping. Before a graceful shutdown, Makalu also reinstates all objects in medium object freelists to the corresponding block in a similar manner.

All persistent updates to mark bits are guaranteed to be visible in NVRAM by the time Makalu gracefully stops, using techniques described in § 8.1. This guarantee is sufficient for mark bits to be reliably used in the next online execution cycle (to create the reclaim list) following a graceful shutdown.

A failure will momentarily leave objects in transient freelists unaccounted for. Therefore, Makalu starts with a clean slate in recovery mode and reconstruct a set of mark bits for each block, purely based on object reachability, by using GC. The objects residing in freelist prior to a failure would appear allocated but unreachable in the heap, and are guaranteed (modulo GC conservatism) to be discovered by the offline GC. Hence, the GC's presence enables us to cheaply maintain frequently updated freelists in transient memory.

If the reclaim list for a particular size is empty, the allocating thread allocates a new block (see below) from the chunk freelist. All objects from the newly allocated block are then added to the empty freelist after setting the corresponding mark bits in the header.

6.2 New Block Allocation

To allocate a new block, Makalu searches for the smallest chunk (c_s) (ideally a chunk with a single block) available among the transient chunk freelists. Recall that the chunk freelist is segregated by the chunk size (number of blocks in them). It removes c_s while holding a lock for the specific freelist where it finds c_s. Next, Makalu obtains the header for c_s. A block is allocated from c_s using the following steps:

1. Remove the first block (b_1) from c_s by adjusting the chunk's size (`hb_sz`) and the beginning address (`hb_block`) in its header.
2. Allocate a header for b_1 and add it to the header map.
3. Set b_1 header fields `hb_block`, `hb_sz` with the starting address, and the size of the object to be allocated from b_1 respectively. Set the `hb_flag` indicating currently in use.

Once the block is allocated, the remaining portion of c_s is returned to the chunk freelist appropriate for its new size.

Failure-induced partial NVRAM updates from steps 1-3 above can cause inconsistencies in the core persistent metadata, leading to undesirable effects, such as a permanently leaked block. Therefore, the allocating thread performs all persistent updates to core metadata in steps 1-3 with ACID guarantees (see § 8 for failure atomicity).

6.3 Large Object Allocation

Allocation requests for large objects are rounded up to the nearest multiple of a block and serviced directly from the chunk freelist. The process is similar to allocating a new block described in § 6.2.

6.4 Expansion of Heap Space

The allocating thread expands the heap when the chunks are insufficient to fulfill an outstanding memory request. Heap expansion requires performing the following steps with ACID guarantees because updates to core metadata are involved.

1. Increment the NVRAM region bump pointer, rgn_curr (stored as base metadata, see § 5.5)
2. Allocate a header for the acquired chunk and add it to the header map
3. Store chunk's size, and the starting address in the header, and set the flag in the header

Once the heap is expanded, allocations can occur as described earlier.

7. Deallocation

Using the header map (§ 5.2), the deallocating thread computes the block and the size of the object from the address being deallocated. If it's a small object, it's added to the start of the deallocating thread's local freelist corresponding to its size. It does not require any update to persistent metadata. We do not attempt to return the object to the allocating thread. This is in contrast with the approach taken by some transient allocators such as Hoard [9] to prevent an allocator from inducing false cache line sharing in multi-threaded applications. Similar to [27], we expect a programmer to allocate cache-aligned objects where false sharing is a concern. Medium-sized objects are added to the corresponding global freelist after acquiring the per size lock (see § 6).

7.1 Object Freelists Truncation

The total bytes of memory held in each medium and small object freelist is capped at twice the size of the block. If a transient freelist (for small and medium objects) grows to its maximum capacity when a thread deallocates an object, the thread is responsible for truncating the freelist by half, leaving the top half of the objects for future allocation requests. This design has the following advantages:

- It prevents unbounded memory blowup in applications with producer-consumer de-/allocation patterns among threads [9].
- It bounds the amount of work that a gracefully terminating thread has to perform to purge its small object freelist. It also bounds the number of medium objects in the global freelist that Makalu has to process before a graceful shutdown.

To truncate a free list, a thread removes one object at a time from the freelist, looks up the block header for the ob-

ject removed from the freelist, and marks the object as available in the future for other threads (to add to their freelist) by clearing the corresponding mark bit in the block header. Moreover, if a thread notices that a sufficient number of objects have become free in a block when clearing the mark bit, it adds the block back to the reclaim list. It acquires a lock for the appropriate reclaim list to do so. Note that objects in a single freelist may come from more than one block due to a number of factors such as objects being deallocated from a previous execution cycle, remotely allocated object being added to the local freelist following a deallocation and so on. Consequently, one or more blocks may be put back into the reclaim list.

Other threads can re-use the block returned to the reclaim list to refill their freelists at later times (§ 6). All updates to mark bits are guaranteed to be eventually visible by the time Makalu shuts down gracefully see (§ 8.1). If a failure occurs before all mark bits become visible, GC will start with a clean slate and create a consistent set of mark bits in recovery mode as described in § 6.1.

7.2 Empty Block Deallocation

It is quite possible for the entire set of mark bits to be cleared for some block during the freelist truncation. A set of all clear mark bits indicates that objects belonging to the block are neither allocated nor do they reside in any of the transient freelists. At this point, it is safe for Makalu to deallocate the entire block and add it back to the chunk freelist. This enables the block to be re-used to fulfill memory requests for another size class. Figure 5 shows the pseudocode for block deallocation. The deallocating thread attempts to coalesce with the immediately preceding (lines 9–20) or the following block/chunk (lines 21–30) (if they exist and are free) to create a larger chunk of memory in the heap. All updates to the core metadata in header fields are performed using an internal interface (`store_nvm_*`) within an all-or-nothing code section demarcated by `start_nvm_atomic` (line 8) and `end_nvm_atomic` (§ 8) (line 36). If a failure occurs before the complete set of updates to the header metadata in method `deallocate` becomes visible in `NVRAM`, partial updates are guaranteed to be undone restoring the consistency of all headers involved. The GC will subsequently rediscover the completely empty block offline and will deallocate it using the same fail-safe approach.

```
1: deallocate(Block* b)
2: {
3:  lock(gml);
4:  hdr* cHdr = map.find(b);
5:  hdr* pHdr = map.find(b-1);
6:  hdr* nHdr = map.find(b+1);
7:  /* ACID section */
8:  start_nvm_atomic();
9:  /* coalesce with previous block */
10: if (pHdr && isFree(pHdr)) {
11:  removeFromChunkFL(pHdr -> hb_block);
12:  /* ACID update, core metadata */
13:  store_nvm_word(&pHdr -> hb_sz,
14:    pHdr -> hb_sz + BLOCK_SZ);
15:  store_nvm_word(&cHdr -> hb_sz, 0);
16:  store_nvm_addr(&cHdr -> hb_block,NULL);
17:  uninstall(cHdr);
18:  cHdr = pHdr;
19:  coalesced = 1;
20: }
21: /* coalesce with next block */
22: if (nHdr && isFree(nHdr)){
23:  removeFromChunkFL(nHdr->hb_block);
24:  store_nvm_word(&cHdr->hb_sz,
25:   cHdr->hb_sz + nHdr->hb_sz);
26:  store_nvm_word(&nHdr->hb_sz, 0);
27:  store_nvm_addr(&nHdr->hb_block, NULL)
28:  uninstall(nHdr);
29:  coalesced = 1;
30: }
31: /* update the current hdr */
32: if (!coalesced){
33:  store_nvm_word(&cHdr->hb_sz, BLOCK_SZ);
34:  store_nvm_int(&cHdr->hb_flag, FREE);
35: }
36: end_nvm_atomic();
37: addToChunkFL(cHdr);
38: unlock(gml);
39:}
```

Figure 5: Pseudocode for empty block deallocation

7.3 Large Object Deallocation

Large object deallocation returns such objects directly to the appropriate chunk freelist using steps similar to those described above for empty allocated blocks.

8. ACID Guarantees for Metadata

Makalu uses internal interfaces presented in figure 6 to specify a set of persistent stores (to core metadata) that must be atomic w.r.t failure, i.e. which need to be visible in `NVRAM` on an all-or-nothing basis. Makalu uses undo-logs to enforce failure atomicity guarantees. `start_nvm_atomic` (lines 1–3) marks the beginning of the set of failure-atomic writes. A call to this method is always preceded by an acquisition of the global mutex lock which provides the isolation guarantees amidst multiple mutator threads. Within the atomic section, core metadata is modified using one of the `store_nvm_*` methods (lines 8–26) based on the metadata type. Details for the integer method are shown in the figure (lines 8–23). Each of these methods creates a log entry containing the memory address, the current value of the metadata , and its data type (lines 9–12) before storing the new value. The new log entry is published (lines 14–16) by stamping the entry with the current value of `md_log_version`. Recall that the current value of

```
1:  start_nvm_atomic(){
2:   next = md_log_space_start;
3:  }
4:  end_nvm_atomic(){
5:   md_log_version++;
6:   FLUSH(md_log_version);
7:  }
8:  store_nvm_int(int* addr, int val){
9:   /* create a log entry */
10:  next->addr = addr;
11:  next->val.int_val = *addr;
12:  next->type = INT;
13:  MEMORY_FENCE();
14:  /* publish */
15:  next->version = md_log_version;
16:  FLUSH(next);
17:  /* next log entry pos. */
18:  next++;
19:  /* store the value */
20:  *(addr) = val;
21:  /* make the update visible */
22:  FLUSH(addr);
23: }
24: store_nvm_word(int* addr,int val);
25: store_nvm_char(int* addr,char val);
26: store_nvm_addr(void** addr,void* val);
```

Figure 6: Internal facility for failure-atomic updates

`md_log_version` is stored as Makalu's core base metadata (see § 5.5). Makalu ensures that the log entry is visible in `NVRAM` using memory fences (line 13) and cache line flushes (line 16) before storing the new value and making the value visible in `NVRAM` (lines 19–22). A call to `end_nvm_atomic` marks the end of the failure-atomic updates by incrementing the `md_log_version` and flushing it to `NVRAM` (lines 4–7).

If a failure occurs before the incremented value of `md_log_version` becomes visible in NVRAM, Makalu starts in an offline phase. It then infers that there are partial updates from the previous run if there is a log entry in the designated log space with the same version as `md_log_version` (the current log version visible in `NVRAM`). The last log entry with that version is obtained and the undo entries are applied in reverse order of their creation, thus nullifying the effects of original updates. Once it has fully replayed the relevant log entries and flushed all the effects of replay to `NVRAM`, it increments the log version (while still offline) and makes it visible in `NVRAM`. The number of log entries never grows beyond a certain number (20 entries) because the provided interface is only used internally in specific scenarios. Each scenario has a statically known number of related updates.

8.1 Eventual Visibility for Metadata

The consistency of auxiliary metadata such as header map and mark bits is guaranteed using a technique less aggressive than one described above. Multiple updates to these metadata can be allowed over the course of online execution before issuing a cache line flush to ensure their visibility. All updates only need to be visible by the time of the graceful shutdown to avoid expensive recovery and restart. We explicitly guarantee their visibility before the graceful shutdown to avoid the chance of dirty cache lines getting lost because of a hardware failure between graceful process termination and next restart. We use a scheme, roughly analogous to one used in [14], that tracks multiple updates to the same cache line using a fixed-size hash table. Modified cache lines that have not yet been flushed appear in the table. Hash table collisions are resolved by flushing the cache line corresponding to the previous entry and removing it. After the last update to auxiliary metadata before the graceful shutdown, the hash table is emptied by flushing each dirty cache line being tracked in the table.

9. Offline Recovery and GC

The offline phase has distinct steps that must be initiated in the following sequence:

9.1 Recovery

Following a failure, Makalu starts in an offline mode. The log replay ensures that the core metadata is restored to a consistent state. Next, Makalu purges the existing header map and builds a new one by scanning each header space and adding headers currently in use to the header map. Mark bits in the header are also cleared in the process. Free headers found (not currently assigned to any block) are added to HFL (§ 5.1).

9.2 Garbage Collection

The `NVMPL` with which the allocator has been integrated typically makes a request to collect garbage once it has restored the user data in the heap to a harmonious state. Such garbage collection is significantly simpler than a conventional garbage collector since there is no mutator present, and the set of garbage collection roots is limited to a small set of explicitly specified region roots. The effort that garbage collectors normally invest in, for example, stopping threads and parsing thread stacks, is unnecessary. So are the usual constraints on the compiler to avoid concealed pointers in registers, etc.

It would be possible to restrict `NVRAM` data structures so that pointer locations in `NVRAM`-allocated objects are apparent. We could require that for `NVRAM` data structures, roots and pointer fields be declared with their correct type (and not, for example, as `void *` or `intptr_t`), and that unions be suitably restricted. We expect to adopt at least some such restrictions or annotations in the future so that the garbage collector can reliably clear persistent-to-transient pointers. Such pointers are obviously invalid after a process restart.

For now, we instead adopt the parallel mark and sweep algorithm from bdwgc as described in [12, 13] in a fully con-

servative mode for our C/C++ setting. It imposes the fewest restrictions on reuse of existing code in an NVRAM setting. Although this clearly affects the details we describe below, we do not believe it affects the fundamental benefit of greatly reducing the cost of metadata updates during allocation.[3]

The mark phase starts by analyzing persistent roots explicitly registered as part of the NVM region (see § 5.4).

The entirely empty blocks found at the end of the mark phase are deallocated using the process described in § 7.2. All mark bits for each partially allocated blocks are made visible in NVRAM before the offline recovery completes. This enables the reliable recreation of reclaim list at the beginning of the online restart and threads to subsequently refill object freelists using blocks in the reclaim list (i.e. perform online incremental sweeping; see § 6.1).

10. Integration with NVMPL

10.1 NVMPL Facing Interfaces

Makalu's public interface (Table 1) provides an easy out-of-the-box integration with NVMPLs. We expect an NVMPL to enable Makalu to set itself up in a new NVRAM region using the one-time call to MAK_start. Each time an NVRAM region is reused online, we expect an NVMPL to reinitialize Makalu via MAK_restart. Makalu restarts from the metadata stored in NVRAM region and gets ready for de-/allocation requests by rebuilding various transient internal lists such as reclaim lists and chunk freelists from persistent metadata. As a part of closing the region, we expect the NVMPL to call MAK_close to signal Makalu to shutdown gracefully. This involves taking down transient freelists and guaranteeing visibility of all persistent metadata updates.

For reasons discussed in § 3.4, Makalu may need to defer deallocation requests made using MAK_free. Makalu allows the NVMPL to set up a deallocation callback method (which is invoked by Makalu each time MAK_free is called) via MAK_set_free_cb. For each object reported by Makalu through the callback, the NVMPL can use a secondary MAK_safe_free method to actually deallocate when it is safe to do so. This approach neatly hides the complexity of deferred deallocation from the programmer.

After a failure, the NVMPL is expected to start the recovery phase by using MAK_start_off. At the end of this method call, Makalu would have recovered its metadata to a consistent state. Once the user data is in a harmonious state, the NVMPL can invoke GC using MAK_collect.

[3] Note that the "black-listing" mechanism in bdwgc is not currently functional in this setting, since we have no data from prior GCs about misidentified pointers to unallocated memory. We expect that, due to the small root set size, it is also much less necessary than usual. It could be restored with a fully concurrent, approximate, trace phase while the program is running. Since this would not be used to reclaim memory, it could be allowed to err in both directions. There should not be any need to synchronize with the mutator. Or we could just broaden the scope of garbage collection.

10.2 Integration with Atlas and Mnemosyne

We substituted the default allocators found in two published NVMPLs, Atlas [14] and Mnemosyne [39], with Makalu. These two NVMPLs differ in their failure consistency semantics. Atlas infers durable critical sections from lock-based code, whereas Mnemosyne builds support for persistence around software transactional memory for persistence.

The integration was straightforward for the most part. Deferring deallocation was the most challenging aspect of integration in both cases. Both set up a deferred callback method at the beginning of each execution cycle. In Mnemosyne, deallocations within a transaction have to be deferred. Mnemosyne creates a list of objects which are deallocated within the transaction (as reported by Makalu using callback) and deallocates them as a post-commit action using MAK_safe_free method.

Similar to Mnemosyne, Atlas creates a list of objects deallocated (reported by Makalu) within a failure-atomic section (FASE) and associates the list with the current FASE. The logs associated with a FASE are pruned by a distinct helper thread in Atlas [14]. When the helper determines that a certain FASE can be pruned, it deallocates all objects in the list associated with the FASE using MAK_safe_free method. In both Mnemosyne and Atlas, a programmer only interacts with the standard malloc/free interface, while the complexity of deferred deallocation is completely hidden from them.

Although we chose log-based NVMPLs to evaluate our approach, Makalu works with other forms of NVMPLs, say an NVMPL based on a copy-on-write approach for failure consistency. Note that Makalu's internal metadata or its internal log is never exposed to the programmer or NVMPL and has absolutely no relation with the log generated by a log-based NVMPL. Programmers and NVMPLs must concern themselves with only Makalu's API listed in table 1.

11. Execution Stages and Failure Mitigation

Makalu is designed to handle tolerated failures at all stages of its execution, offline and online. In this section, we summarize Makalu's expected behavior in the case of a failure at various stages of execution.

11.1 One-Time Online Initialization

The initialization is considered complete once the call to the method MAK_start returns. At the end of the method call, Makalu flushes all persistent metadata updates to NVRAM. Next, it stamps the NVRAM region with a "magic number" and flushes this update synchronously to NVRAM before the method returns. Each time Makalu restarts, it checks for this magic number to ensure that the given NVRAM region has been initialized properly. If a failure occurs before this number is visible in NVRAM, Makalu requires the client NVMPL to re-run the initialization routine. If a failure occurs after a successful initialization and before the first call to de-/allocation routines, Makalu's persistent metadata remains unaffected

and hence, a client NVMPL may optionally skip Makalu's offline recovery routine altogether.

11.2 Online Re-initialization

Recall that a client NVMPL uses method call MAK_restart to re-initialize Makalu from its persistent metadata each time NVRAM region is reopened for access. Makalu only reads the persistent metadata to recreate transient structures during this stage. Hence, a failure does not affect Makalu at all at this stage. A client NVMPL can optionally skip Makalu's offline recovery routine in this case as well.

11.3 Online Execution

Any failure after the first call to de-/allocation routines and before the call to MAK_close can corrupt Makalu's metadata and lead to persistent memory leaks. In this case, a client NVMPL is expected to restart Makalu offline and invoke recovery/GC routine.

11.4 Offline Recovery

Recall that Makalu's offline recovery has the following three distinct steps. A failure before the completion of all three recovery steps requires Makalu to be restarted in offline recovery mode.

Step 1: Persistent log replay. First, Makalu replays outstanding persistent undo logs (if present) to restore the consistency of core metadata (see § 8). After Makalu flushes all the updates from log replay to NVRAM, it increments the log version and flushes it synchronously. If a failure occurs before the new incremented log version is visible in NVRAM, it detects the same outstanding logs and replays them again.

Step 2: Reconstruction of auxiliary metadata. Recall that Makalu recreates auxiliary persistent metadata from consistent core metadata. If a failure occurs after the successful completion of step 1 and before the completion of step 2, the recovery resumes by discarding the partially constructed auxiliary metadata, reclaiming the metadata space and reconstructing the auxiliary metadata in that space.

Step 3: Garbage collection. If a failure occurs during a GC or anywhere else after step 2 and before the call to MAK_close returns, Makalu discards partially constructed mark bits and restarts the mark phase with a clean slate. When the call to MAK_close returns, a complete set of mark bits are guaranteed to be visible in NVRAM and the recovery is complete.

12. Related Work

Our work is most closely related to nvm_malloc and pmem [2, 36][4]. nvm_malloc differs from our work in two distinct ways. In nvm_malloc, memory allocation requires two method calls. The first only reserves persistent memory of the requested size. The second method takes the returned persistent address, together with at least one persistent result location, so that it can failure atomically perform the actual allocation and store (publish) the allocated memory address in the provided location. This is thus not a drop-in replacement for conventional malloc and pays a higher cost than Makalu for every allocation. Finally, nvm_malloc does not address interfacing with other NVMPLs. It does not provide a mechanism to defer deallocations. Hence, it is not clear to us how it operates correctly with NVMPLs such as Atlas [14] and Mnemosyne [39], which use a log-based approach to support failure-atomic updates of persistent data.

None of the existing NVMPL allocators provide a safety net against programmer-induced memory leaks. Makalu's offline garbage collection works as this safety net. NV-Heaps [15] is the only published NVMPL which offers anything comparable, by providing a reference counting GC. However, it requires a programmer to distinguish between a strong and a weak reference to break cycles in a persistent heap. We argue that this is error-prone, especially when dealing with large and complex data structures. NV-Heaps is not publicly available and the paper does not indicate that GC was used as a means of reducing the NVRAM allocation cost, nor is it clear this could be done.

A plethora of transient memory allocators have been developed to satisfy the scalability needs of multicore computing [9, 18, 21, 22]. Our work differs from these as our core objective is low cost *crash-resilient persistent* memory management in addition to multi-core scaling. The next section does compare our allocator with a commercially used transient allocator, Hoard [9].

Our offline garbage collection is an adaptation of the conservative garbage collection algorithm implemented in the bdwgc collector [12, 13]. However, garbage collection occurs only offline, when top-level roots are precisely known. Bdwgc does not support NVRAM allocation, and its free() implementation does not scale on multiprocessors.

13. Evaluation

Although NVRAM is getting closer to becoming widely available [1, 31], it is not at the moment. Hence, we used Linux *tmpfs*[37] to simulate NVRAM during the collection of results. As files in *tmpfs* are only backed by DRAM, we used a process crash and restart to test Makalu's crash resilience and restart logic. In order to measure failure consistency overhead, we did enable full failure consistency mechanisms such as cache line flushes and memory fences, as we would have in actual NVRAM systems.

Unless otherwise stated, we compiled all the code used in this evaluation using the GNU gcc/g++ compiler, version 4.8.4 at optimization level "-O2". We ran our experiments on a 64-bit Ubuntu machine (kernel version 3.13.0-66-generic) that has 12 GiB of RAM, two Intel Xeon E5-2695 processors, 6 cores per socket (12 cores total), and hyper-threading switched off. All results collected were averaged over 6 runs.

[4] Both of these work essentially use a similar two-step allocation algorithm that we henceforth refer to as nvm_malloc.

13.1 Comparison with Existing Allocators

We compare the allocation speed, throughput, and multi-threaded performance of our allocator with another persistent allocator, ***nvm_malloc***, which is built upon jemalloc [36]. Although our goal is not to produce the fastest or the most scalable transient allocator, we compared our allocator with ***Hoard***(ver3.10)[9], one such popular allocator, to show that Makalu's raw performance is comparable despite the extra failure consistency overhead it has to incur. We also include in our comparison the following three versions of bdwgc:

1. An unmodified version (ver7.2) with multi-threaded GC, thread-local allocation enabled, and explicit deallocation disabled (***bdwgc***).
2. Same as (1), but with GC disabled and the default sub-optimal explicit deallocation enabled (***bdwgc-free***).
3. A modified version of (1), with a better support for explicit deallocation and GC disabled (***bdwgc-mod***).

We use the following often used allocation benchmarks for comparison.

Larson: This benchmark has often been used to simulate a multi-threaded long running server [6, 9, 12, 27, 28]. We configured the benchmark to run for 10 seconds, with t threads where each thread runs for 10^4 rounds, allocating and deallocating 10^3 64-byte[5] objects in each round. It reports the allocation throughput in a given time window in terms of operations/sec.

Threadtest: This benchmark has been used by [6, 9, 27] to measure multi-threaded scalability performance of an allocator. Each thread allocates and deallocates memory in a tight loop with a configurable amount of work in-between. For t threads, we ran the benchmark such that each thread performed 10^4 rounds of de-/allocation, and $\frac{10^5}{t}$ de-/allocations in each round.

Prod-con: The benchmark simulates applications which have two mutually exclusive sets of allocating and deallocating threads. Such de-/allocation pattern typically causes memory blowups [9] or performance degradation in poorly designed allocators. The benchmark starts an even number of threads t, and creates $\frac{t}{2}$ blocking queues [32]. It then assigns a producer-consumer thread pair to each queue. An object is allocated by a producer thread, passed to a consumer thread via a queue, and deallocated there. Each pair of producer-consumer threads de-/allocates $\frac{2\times10^7}{t}$ objects that are 64 bytes in size.

Results: The multi-threaded performance results presented in figure 7 shows that Makalu performs orders of magnitude better than `nvm_malloc` in common memory allocation patterns presented by above benchmarks. The difference in performance between Makalu and bdwgc-mod is remarkably thin, making it safe to say that Makalu is only slightly burdened by failure safety of its metadata. We compare the failure consistency overhead of Makalu with `nvm_malloc` in terms of the average number of cache line flushes issued per thread and present results in figure 8 for all three benchmarks.

Makalu memory consumption (transient + persistent) is somewhere between the best and worst case observed and is comparable to that of `nvm_malloc` across all three benchmarks. This is especially true as the thread count increases. We do present full memory consumption results in appendix A.

Although our evaluation assumes the read/write latency for `NVRAM` and `DRAM` to be comparable (as has often been predicted), we believe that the advantage shown here of our allocator over existing persistent allocators w.r.t. to allocation speed, multi-core scalability, and failure consistency overhead will continue to hold even in the case where `NVRAM` has greater access latency than `DRAM`. Our low footprint for core persistent metadata is the primary reason for our low failure consistency overhead. This indicates we have a comparatively low number of unavoidable reads/writes to `NVRAM`.

13.2 Recovery and Garbage Collection

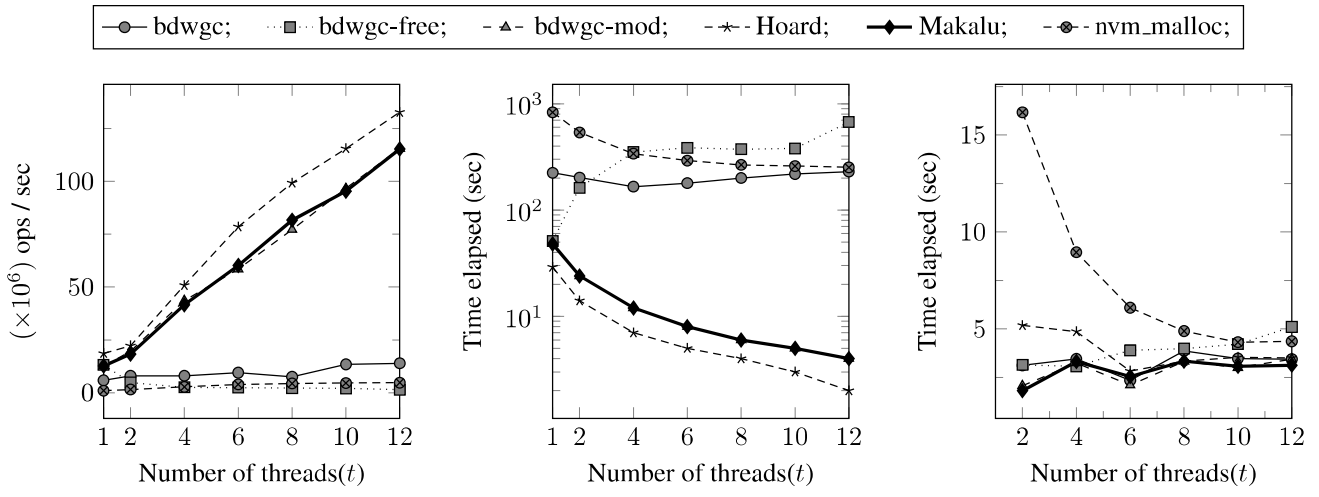
SZ *MiB*	MX_L $\times10^3$	T_{ON} *sec*	T_{REC} *ms*	T_{GC} *ms*	$OVER_{off}$ %
200	3.39	1.01	4.18	60.98	6.45
400	6.66	2.00	6.77	133.97	7.05
600	9.84	2.97	7.36	154.94	5.46
800	13.06	3.94	11.73	276.90	7.33
1,024	16.67	5.04	13.66	349.88	7.22
2,048	32.82	10.20	28.23	699.23	7.13
3,072	49.48	15.08	37.96	1,030.57	7.08

Table 2: Makalu offline recovery and GC performance. SZ = size of allocated heap before crash; MX_L = longest pointer chain explored during GC; T_{ON} = online execution time before the crash; T_{REC} = time taken to recover allocation metadata and restart offline; T_{GC} = time consumed by offline GC; and offline overhead, $OVER_{off}$ = the ratio of time taken offline to time taken online to fill-up heap, $\frac{T_{REC}+T_{GC}}{T_{ON}} \times 100$.

We achieve low online failure consistency overhead in Makalu at the cost of offline recovery and GC in the rare event of failure. In this section, we quantify this offline cost using the following benchmark.

Resur: The benchmark allocates SZ *MiB* of persistent memory in total by allocating persistent objects of random size up to half a page before the benchmark crashes abruptly using process abort. For most environments, this models an absurdly short interval between failures. With each allocation, a coin is tossed to either retain the allocated object, in which case it is made reachable from one of 512 `NVRAM` re-

[5] This is the smallest common allocation size; `nvm_malloc` only supports allocations in the multiples of 64 bytes

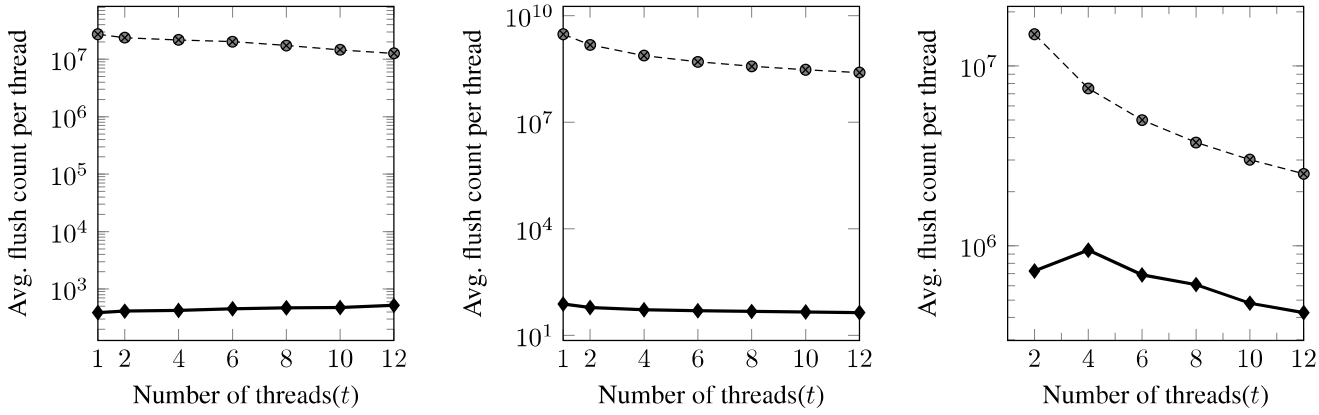


(a) Larson: allocation throughput (higher is better)

(b) Threadtest: time elapsed vs. thread count (lower is better)

(c) Prod-con: time elapsed vs. thread count (lower is better)

Figure 7: Allocation benchmarks: throughput and multi-threaded performance



(a) Larson (lower is better)

(b) Threadtest: (lower is better)

(c) Prod-con (lower is better)

Figure 8: Allocation benchmarks: failure consistency overhead (Makalu vs. `nvm_malloc`)

gion roots (randomly selected) or deallocated immediately. The retained objects essentially form a collection of linked lists of variable length rooted at region roots. At the time of the crash roughly $SZ/2$ of the memory is reachable in the persistent heap. Following a crash, Makalu is started in offline mode. It first recovers its allocation metadata to a consistent state and then performs a parallel GC.

Results: Table 2 shows that the time to restart/recover metadata as well as the time to collect garbage (reported in 4^{th} and 5^{th} columns respectively) are quite small and grow modestly with the total size of the heap. Offline overhead data in the 6^{th} column shows that the total time spent in offline recovery and garbage collection remains a small and somewhat constant fraction of the time spent in online allocation (3^{rd} column) even as the total size of the allocated memory (1^{st} column) and the maximum length of the pointer chain in the heap (2^{nd} column) grows.

We believe that this result greatly enhances the possibility of Makalu being profitably used in all but most peculiar cases where applications have very high rates of failure. In such cases, persistent applications may have more pressing problems than efficient memory allocation.

13.3 Comparison with NVMPL Default Allocators

In this section, we compare the performance of Makalu to that of the existing `NVMPL`s' default allocators. The default allocator in Mnemosyne is built upon Hoard [9, 39].

The possibility of in-place persistence of data became real only recently with the promise of `NVRAM` in the near future. As such, to the best of authors' knowledge, the earliest `NVMPL` work [15, 39] is barely half a decade old and still

immature. Our work fills in a prior omission by providing a leak-free memory allocator with programmer friendly interface. It aims to promote the development of new NVRAM applications and a more programmable NVMPL, but it still somewhat suffers from the status quo of not having standard real NVRAM application benchmarks. Given the circumstances, we use the following three applications, which could benefit from in-place persistence in the future. Furthermore, other transient memory management papers [9, 17] have often made use of some of these benchmarks.

Barnes-Hutt: This benchmark is a multi-threaded N-body problem solver [8]. The initial set of 100,000 particle positions and forces are stored in NVRAM, as are the resultant particle positions and forces after each time-step.

N-queens: It is a lock-based, multi-threaded implementation of a recursive search algorithm for finding a solution to the n-queens problem [10]. It uses a pool of workers and work queues. Both work queues, as well as units of work, are allocated in NVRAM. A unit of work is essentially a search frontier to be further explored. Each worker removes a unit of work from the queue and pushes new work generated back to the queue. We explored the solution for the 16-queens problem using a variable number of worker threads. Each worker has to acquire a lock to add or remove work from the queue.

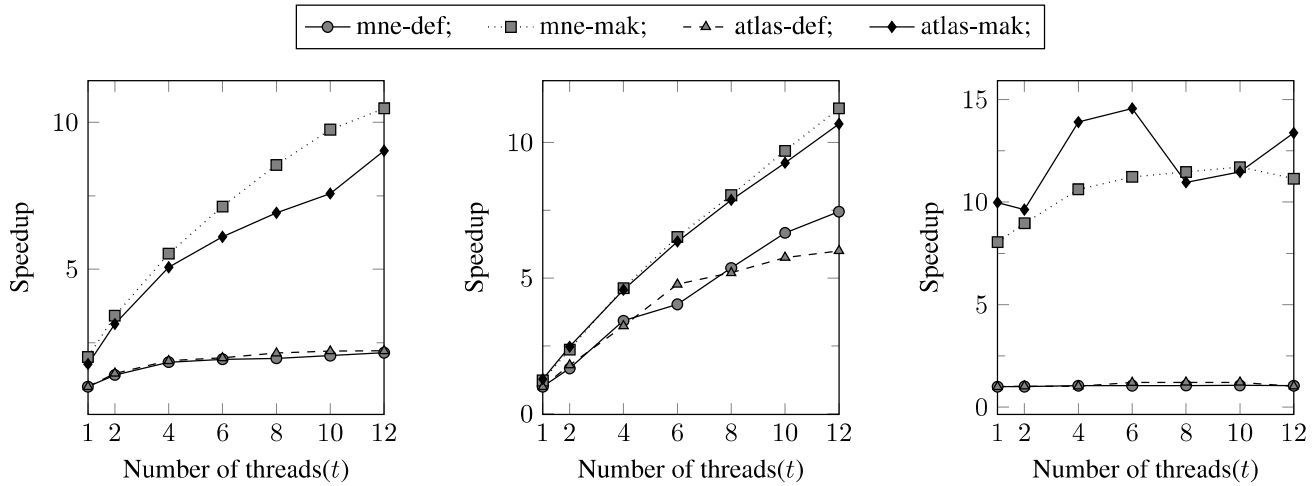
Cholesky: It is a multi-threaded, tile-based algorithm for decomposing a dense matrix into a lower triangular matrix and its conjugate transpose [20]. Using a variable number of threads, we allocated a 1000×1000 input matrix in NVRAM, decomposed it using tile size 4×4 and stored results in NVRAM as well.

We integrated the same version of the Makalu allocator with the Mnemosyne and the Atlas library. We shall refer to the Makalu-integrated Mnemosyne and Atlas as ***mne-mak*** and ***atlas-mak***, whereas versions with default allocators as ***mne-def*** and ***atlas-def*** respectively. Mnemosyne requires special Linux kernel support and compiler support. Therefore, Mnemosyne was compiled using Intel compiler prototype edition 3 with -O2 optimization and results were obtained on a machine (same architectural specification as above) running Centos 2.6.32 Linux distro.

	Barnes-Hutt *sec*	N-queens *sec*	Cholesky *sec*
atlas-def	16.44	23.72	19.67
mne-def	18.10	19.69	13.98

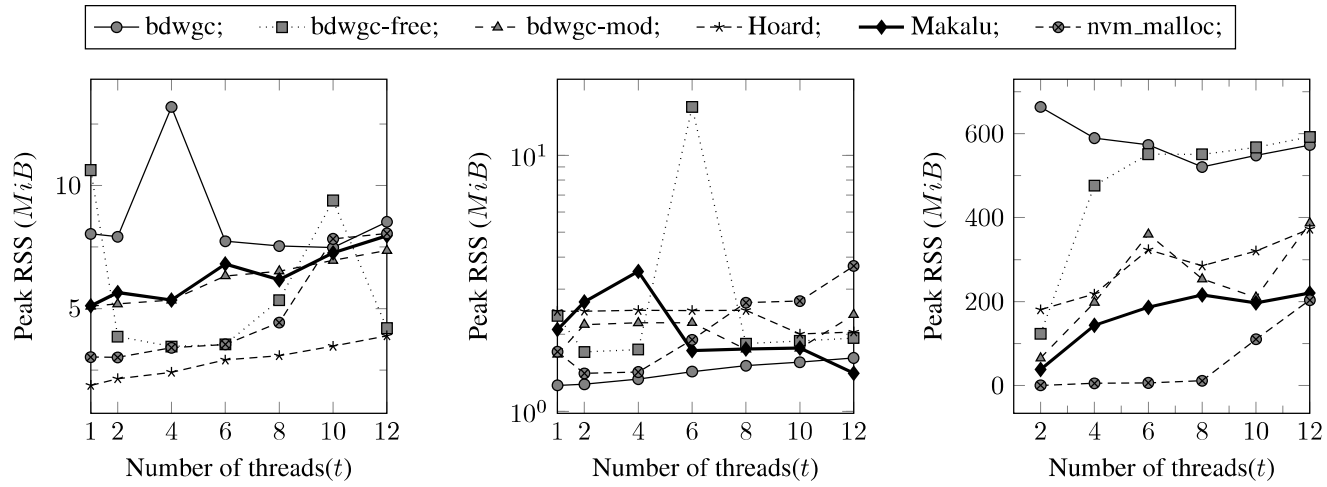
Table 3: Single-threaded base performance for Atlas and Mnemosyne default allocators.

Results: Figure 9 compares the default allocators in each NVMPL to Makalu using speedups obtained on the above benchmarks. Since Atlas and Mnemosyne data are collected on two different systems, they should not be compared to each other. Furthermore, Atlas and Makalu speedups are calculated based on the single-threaded performance of their respective default allocators on above benchmarks on their respective machines. Table 3 summarizes these base values. In all three benchmarks, the Makalu-integrated version of NVMPL outperforms the default version by orders of magnitude. The result also demonstrates the easy interoperability of Makalu with more than a single NVMPL. In order to isolate and compare the allocation cost, we only concerned ourselves with avoiding inconsistencies in heap metadata for these sets of results. We are not concerned with maintaining the consistency of the data stored in persistent heap itself. For one such NVMPL, namely Atlas, more extended results



(a) Barnes-Hutt (higher is better) **(b)** N-queens (higher is better) **(c)** Cholesky (higher is better)

Figure 9: Comparison of Makalu with NVMPLs' default allocators. Speedup is computed w.r.t. single threaded performance of respective NVMPLs' default allocators. Refer to table 3 for single-threaded base timing values.

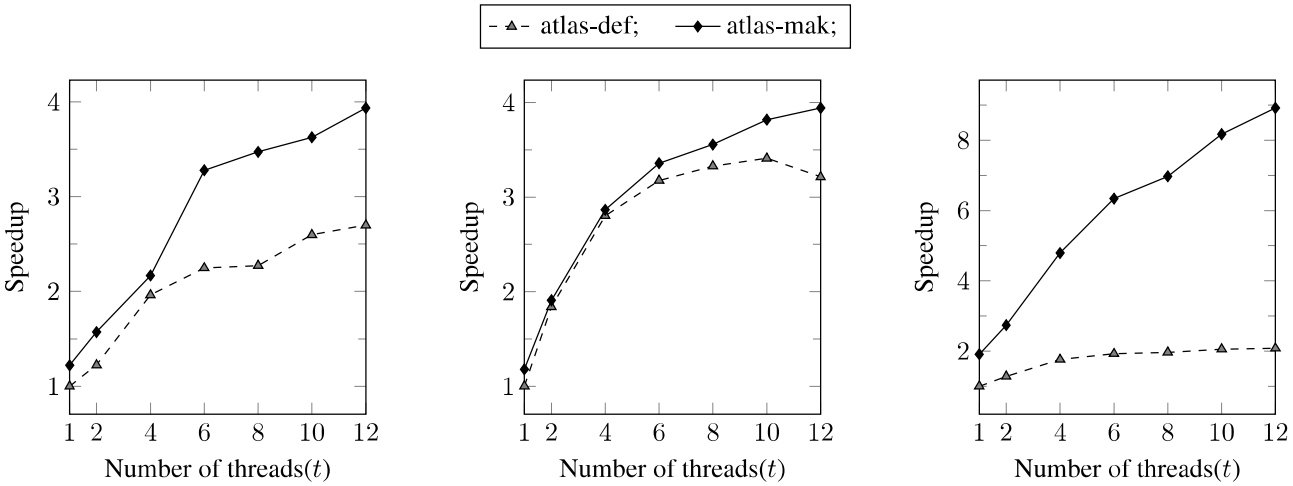


(a) Larson: Peak RSS vs. thread count (lower is better)

(b) Threadtest: Peak RSS vs. thread count (lower is better)

(c) Prod-con: Peak RSS vs. thread count (lower is better)

Figure 10: Allocation benchmarks: peak memory consumption



(a) Barnes-Hutt (higher is better)

(b) N-queens (higher is better)

(c) Cholesky (higher is better)

Figure 11: Comparison of Makalu with the Atlas' default allocator: full failure consistency enabled. Speedup computed w.r.t. single threaded performance of Atlas default allocator. Refer to table 4 for the base timing values.

for above benchmarks with full failure consistency mechanisms enabled are presented in appendix A.

14. Conclusions

We have shown that it is possible to build a memory allocator for `NVRAM` that maintains the standard malloc()/free() programming model, correctly ensures persistence of metadata, and interoperates with multiple `NVRAM` persistence libraries. Surprisingly this is possible at a cost comparable to transient memory allocators.

Our crucial observation is that by relying on offline garbage collection during failure recovery, the per allocation persistence overhead is greatly reduced. A typical small object persistent allocation does not need to flush any data to persistent memory since all relevant metadata can effectively be reconstructed from the object graph during recovery.

Acknowledgments

The first author was partially supported by the US Department of Energy under Cooperative Agreement no. DE-SC0012199. We thank the OOPSLA'16 anonymous referees for their valuable comments. Our work has benefited immensely from our discussions with many people at Hewlett Packard Labs including Milind Chabbi, Charlie Johnson,

Terence Kelly, Hideaki Kimura, Harumi Kuno, Mark Lillibridge, Charles B. Morrey III, and Haris Volos.

A. Extended Results

A.1 Comparison with Existing Allocators: Memory Footprints

Figure 10 presents peak memory consumption of allocators discussed in § 13.1 for benchmarks used in that section. Makalu's peak memory consumption is comparable to commercially available transient allocators such as Hoard. For all three benchmarks, Makalu's memory consumption is comparable to other allocators. `nvm_malloc` had relatively lower peak memory footprint across all benchmarks.

A.2 Comparison with Atlas Default Allocator: Full Failure Consistency Enabled

Figure 11 presents results comparing Makalu integrated with Atlas and Atlas' default allocator, using benchmarks discussed in § 13.3. For this set of results, we enabled the full instrumentation of code, using Atlas' LLVM compiler [14] for ensuring failure consistency of user data, in addition to guaranteeing the absence of memory leaks and consistency of heap metadata in case of a failure.

Results in figure 11 demonstrate that turning on full failure consistency support in Atlas hides some of the gains in allocation speed (that we observed in figure 9) from using Makalu. Nevertheless, Makalu yields superior performance to the default Atlas allocator. Barnes-Hutt and Cholesky use barriers for synchronization among threads whereas N-queens uses pthread mutexes. Results in figure 11 show that Makalu-integrated Atlas yields superior performance for both lock- and non-lock-based code.

	Barnes-Hutt *sec*	N-queens *sec*	Cholesky *sec*
atlas-def	63.59	59.63	66.16

Table 4: Single-threaded base performance for the Atlas' default allocator (full failure consistency enabled).

References

[1] NVDIMM special interest group. URL `http://www.snia.org/forums/sssi/NVDIMM`.

[2] Pmem.io: Persistent memory programming. URL `http://pmem.io/`.

[3] NVM programming technical work group. URL `http://www.snia.org/forums/sssi/nvmp`.

[4] Process integration, devices and structures. *International Technology Roadmap for Semiconductors (ITRS)*, 2013. URL `http://www.semiconductors.org/clientuploads/Research_Technology/ITRS/2013/2013PIDS.pdf`.

[5] NVM direct library, 2015. URL `http://www.oracle.com/technetwork/oracle-labs/open-nvm-download-2440119.html`.

[6] M. Aigner, C. Kirsch, M. Lippautz, and A. Sokolova. Fast, multicore-scalable, low-fragmentation memory allocation through large virtual memory and global data structures. In *Proceedings of the 2015 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '15, New York, NY, USA, 2015. ACM.

[7] M. P. Atkinson, P. J. Bailey, K. J. Chisholm, P. W. Cockshott, and R. Morrison. Readings in object-oriented database systems. chapter An Approach to Persistent Programming, pages 141–146. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1990. ISBN 0-55860-000-0.

[8] J. Barnes and P. Hut. A hierarchical O(N log N) force-calculation algorithm. *Nature*, 324(6096):446–449, Dec. 1986.

[9] E. D. Berger, K. S. McKinley, R. D. Blumofe, and P. R. Wilson. Hoard: A scalable memory allocator for multithreaded applications. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS IX, pages 117–128, New York, NY, USA, 2000. ACM.

[10] B. Bernhardsson. Explicit solutions to the n-queens problem for all n. *SIGART Bull.*, 2(2):7–, Feb. 1991. ISSN 0163-5719.

[11] H.-J. Boehm. A garbage collector for c and c++. URL `http://www.hboehm.info/gc/gcdescr.html`.

[12] H.-J. Boehm. Fast multiprocessor memory allocation and garbage collection. Technical Report HPL-2000-165, Internet and Mobile Systems Laboratory, HP Laboratories, Palo Alto, CA, December 2000. URL `http://www.hpl.hp.com/techreports/2000/HPL-2000-165.html`.

[13] H.-J. Boehm and M. Weiser. Garbage collection in an uncooperative environment. *Softw. Pract. Exper.*, 18(9):807–820, September 1988.

[14] D. R. Chakrabarti, H.-J. Boehm, and K. Bhandari. Atlas: Leveraging locks for non-volatile memory consistency. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '14, pages 433–452, New York, NY, USA, 2014. ACM.

[15] 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 *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, pages 105–118, New York, NY, USA, 2011. ACM.

[16] A. Dearle, G. N. C. Kirby, and R. Morrison. Orthogonal persistence revisited. In *Proceedings of the Second International Conference on Object Databases*, ICOODB'09, pages 1–22, Berlin, Heidelberg, 2010. Springer-Verlag. ISBN 3-642-14680-5, 978-3-642-14680-0.

[17] T. Endo and K. Taura. Reducing pause time of conservative collectors. In *Proceedings of the 3rd International Symposium on Memory Management*, ISMM '02, pages 119–131, New York, NY, USA, 2002. ACM. ISBN 1-58113-539-4.

[18] J. Evans. Jemalloc: A scalable concurrent malloc(3) implementation, 2006. URL `https://github.com/jemalloc/`

jemalloc.

[19] T. Gao, K. Strauss, S. M. Blackburn, K. S. McKinley, D. Burger, and J. Larus. Using managed runtime systems to tolerate holes in wearable memories. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 297–308, New York, NY, USA, 2013. ACM.

[20] J. E. Gentle. *Matrix Algebra: Theory, Computations, and Applications in Statistics*. Springer Publishing Company, Incorporated, 1st edition, 2007. ISBN 0387708723, 9780387708720.

[21] W. Gloger. ptmalloc3: Multi-threaded extension to dlmalloc, 2006. URL `http://malloc.de/en/`.

[22] Google Inc. Tcmalloc : Thread-caching malloc, 2011. URL `http://goog-perftools.sourceforge.net/doc/tcmalloc.html`.

[23] A. L. Hosking and J. Chen. Mostly-copying reachability-based orthogonal persistence. In *Proceedings of the 14th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '99, pages 382–398, New York, NY, USA, 1999. ACM. ISBN 1-58113-238-7.

[24] R. L. Hudson, B. Saha, A.-R. Adl-Tabatabai, and B. C. Hertzberg. Mcrt-malloc: A scalable transactional memory allocator. In *Proceedings of the 5th International Symposium on Memory Management*, ISMM '06, pages 74–83, New York, NY, USA, 2006. ACM. ISBN 1-59593-221-6.

[25] Intel Corp. *Intel Architecture Instruction Set Extensions Programming Reference*, . URL `https://software.intel.com/sites/default/files/managed/b4/3a/319433-024.pdf`.

[26] Intel Corp. *Intel64 and IA-32 Architectures Software Developers Manuals Combined*, . URL `http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html`.

[27] B. C. Kuszmaul. Supermalloc: A super fast multithreaded malloc for 64-bit machines. In *Proceedings of the 2015 ACM SIGPLAN International Symposium on Memory Management*, ISMM 2015, pages 41–55, New York, NY, USA, 2015. ACM.

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

[29] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger. Architecting phase change memory as a scalable dram alternative. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, pages 2–13, New York, NY, USA, 2009. ACM.

[30] K. Lim, J. Chang, T. Mudge, P. Ranganathan, S. K. Reinhardt, and T. F. Wenisch. Disaggregated memory for expansion and sharing in blade servers. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, pages 267–278, New York, NY, USA, 2009. ACM.

[31] C. Mellor. SanDisk, HP take on Micron and Intels faster-than-flash XPoint. *The Register*, October 2015. URL `http://www.theregister.co.uk/2015/10/12/sandisk_and_hp_partner_to_counter_xpoint/`.

[32] M. M. Michael and M. L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '96, pages 267–275, New York, NY, USA, 1996. ACM.

[33] Micron. 3D XPoint Technology. URL `https://www.micron.com/about/emerging-technologies/3d-xpoint-technology`.

[34] M. K. Qureshi, V. Srinivasan, and J. A. Rivers. Scalable high performance main memory system using phase-change memory technology. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, pages 24–33, New York, NY, USA, 2009. ACM.

[35] G. Rodriguez-Rivera, M. Spertus, and C. Fiterman. Conservative garbage collectors for general memory allocators. ISMM '00, pages 71–79, New York, NY, USA, 2000. ACM.

[36] D. Schwalb, T. Berning, M. Faust, M. Dreseler, and H. Plattner. nvm_malloc: Memory allocation for nvram. In *Accelerating Data Management Systems Using Modern Processor and Storage Architectures Workshop, In conjunction with VLDB*, 2015.

[37] P. Snyder. tmpfs: A virtual memory file system. In *In Proceedings of the Autumn 1990 European UNIX Users Group Conference*, pages 241–248, 1990.

[38] D. B. Strukov, G. S. Snider, D. R. Stewart, and R. S. Williams. The missing memristor found. *Nature*, 453(7191):80–83, May 2008.

[39] H. Volos, A. J. Tack, and M. M. Swift. Mnemosyne: Lightweight persistent memory. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, pages 91–104, New York, NY, USA, 2011. ACM.