NVAlloc: Rethinking Heap Metadata Management in Persistent Memory Allocators

Zheng Dang
Zhejiang University
Hangzhou, Zhejiang, China

Shuibing He*
Zhejiang University
Hangzhou, Zhejiang, China

Peiyi Hong
Zhejiang University
Hangzhou, Zhejiang, China

Zhenxin Li
Zhejiang University
Hangzhou, Zhejiang, China

Xuechen Zhang
Washington State University
Vancouver, Washington, USA

Xian-He Sun
Illinois Institute of Technology
Chicago, Illinois, USA

Gang Chen
Zhejiang University
Hangzhou, Zhejiang, China

ABSTRACT

Persistent memory allocation is a fundamental building block for developing high-performance and in-memory applications. Existing persistent memory allocators suffer from suboptimal heap organizations that introduce repeated cache line flushes and small random accesses in persistent memory. Worse, many allocators use static slab segregation resulting in a dramatic increase in memory consumption when allocation request size is changed. In this paper, we design a novel allocator, named NVAlloc, to solve the above issues simultaneously. First, NVAlloc eliminates cache line reflushes by mapping contiguous data blocks in slabs to interleaved metadata entries stored in different cache lines. Second, it writes small metadata units to a persistent bookkeeping log in a sequential pattern to remove random heap metadata accesses in persistent memory. Third, instead of using static slab segregation, it supports slab morphing, which allows slabs to be transformed between size classes to significantly improve slab usage. NVAlloc is complementary to the existing consistency models. Results on 6 benchmarks demonstrate that NVAlloc improves the performance of state-of-the-art persistent memory allocators by up to 6.4x and 57x for small and large allocations, respectively. Using NVAlloc reduces memory usage by up to 57.8%. Besides, we integrate NVAlloc in a persistent FPTree. Compared to the state-of-the-art allocators, NVAlloc improves the performance of this application by up to 3.1x.

CCS CONCEPTS

• **Software and its engineering → Allocation / deallocation strategies**; • **Hardware → Non-volatile memory**.

KEYWORDS

dynamic memory allocation, persistent memory, memory fragmentation

ACM Reference Format:
Zheng Dang, Shuibing He, Peiyi Hong, Zhenxin Li, Xuechen Zhang, Xian-He Sun, and Gang Chen. 2022. NVAlloc: Rethinking Heap Metadata Management in Persistent Memory Allocators. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '22), February 28 – March 4, 2022, Lausanne, Switzerland*. ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3503222.3507743

1 INTRODUCTION

Dynamic allocation of persistent memory is heavily used for building high-performance applications from indexing structures [8, 23, 24, 26–28], transactional memory [13, 19, 20, 39], to in-memory database systems [1, 10, 25, 31]. Memory allocators are usually well-tuned for volatile memory (e.g., DRAM) to achieve low latency, high scalability, and low fragmentation [2, 17, 33]. The adoption of persistent memory (e.g., Intel Optane DIMMs [11]) has made researchers rethink the design and implementation of allocators. The allocators designed for persistent memory need to maintain the salient features of DRAM allocators for high-performance memory management. More importantly, they should enforce crash consistency so that they can safely recover allocated memory objects after failures.

Many allocators have been designed for persistent memory [3, 4, 15, 30, 31, 37]. They need to manage persistent heaps via various types of metadata (e.g., object bitmaps, slab structures, extent headers, and write-ahead logs) for efficiently serving memory allocation and deallocation. In this paper, we name them *heap metadata*. For example, PMDK [11] and nvm_malloc [37] use bitmaps to mark objects that have been allocated. PAllocator [31] uses logs to enforce crash consistency of heap metadata. Updating the heap metadata triggers frequent small writes to persistent memory ranging from 1 bit to 64 B. All of these persistent allocators use a size-segregated algorithm for serving small allocation requests to reduce memory fragmentation.

The existing allocators designed for persistent memory have many issues related to heap metadata management. First, small

*Shuibing He is the corresponding author.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ASPLOS '22, February 28 – March 4, 2022, Lausanne, Switzerland
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-9205-1/22/02…$15.00
https://doi.org/10.1145/3503222.3507743

writes to heap metadata may cause cache line reflushes. A typical size of CPU cache line is 64 B [31]. The size of a bitmap is 8 B in nvm_malloc. When the bitmap is updated repeatedly, the same cache line should be flushed for persistence. The latency of cache line reflush is 7.5x higher than the latency of writes [7]. We observe that the number of cache line reflushes accounts for 40.4%~99.7% of the total number of allocator-induced flush operations in four well-known benchmarks (Section 3.1). Frequent cache line reflush operations cause the degraded performance of persistent memory allocators.

Second, heap metadata of allocators tend to be randomly accessed in persistent memory. Many allocators (e.g., PMDK, PAllocator, and Makalu [3]) subdivide the heap into chunks of fixed sizes (e.g., 4 MB) for ease of management. They maintain bookkeeping metadata in each chunk's header space which is separated from data space to avoid metadata being modified by mistake. This layout causes headers to be distributed over the whole heap space. After serving a sequence of allocation and deallocation requests, allocators have to in-place update headers randomly located in persistent memory. Recent work has shown that persistent memory exhibits much worse random access performance than sequential access performance [39, 40] for small writes. Consequently, serving these small random writes to heap metadata prevents the allocators from achieving optimal performance.

Third, static slab segregation causes persistent memory fragmentation. This problem is intensified in persistent memory because the persistent heap is stored on the DAX file systems in the form of files. They cannot be eliminated by restarting the system. All the allocators for persistent memory use size-segregated slabs for small block allocation. Each slab is a container of multiple free blocks and handles a memory allocation of a particular size class. Slabs assigned to one size class cannot be reused for other size classes even though the slabs are mostly empty and there is no free space in slabs of other size classes [38]. This segregation-induced fragmentation increases memory usage by up to 2.8x for workloads with changing allocation sizes and frequent "delete" operations (Section 3.2).

In this paper, we introduce a fast and fail-safe persistent memory allocator named NVAlloc. Its design emphasizes efficiently eliminating cache line reflushes and small random writes and alleviating slab-induced memory fragmentation in heap metadata management. First, NVAlloc uses an interleaved memory mapping from data blocks to their corresponding heap metadata and interleaved layout of linked lists in thread-local caches to avoid accessing the same CPU cache line repeatedly. Second, because in-place metadata updates cause random accesses in persistent memory with the limited write buffer size [40], we add a persistent bookkeeping log to store updates of small metadata in a sequential pattern. As a result, we completely remove random metadata accesses from the critical path of `malloc()` and `free()`. Third, it supports slab morphing with which blocks in two size classes may be co-located in one slab during the slab transformation. Therefore, the free space in slabs of low memory usage can be well utilized, with 4.5% runtime overhead for slab metadata management. Slab morphing is automatically enabled when a slab is mostly idle but cannot be used to serve requests in other size classes.

NVAlloc currently supports both log-based and garbage-collection-based crash-consistency models[1]. Results on 6 benchmarks demonstrate that NVAlloc improves the performance of state-of-the-art persistent memory allocators by up to 6.4x for small allocations and 57x for large allocations. Using NVAlloc reduces memory usage by up to 57.8%. We also integrate NVAlloc in a persistent FPTree [32]. With NVAlloc, the performance of this application is improved by up to 3.1x compared with the state-of-the-art allocators.

2 BACKGROUND

2.1 Terminology

We first define the commonly used terms in persistent memory allocators. **Extents** are a contiguous sequence of bytes allocated from the persistent heap space directly for serving large allocation requests. **Slabs** are pre-allocated extents in persistent memory and containers of fixed-size free blocks. The slab size is 64 KB in this paper. Small allocations are served using slabs based on their size classes. **Blocks** are a contiguous sequence of bytes in persistent memory allocated from the slab structure for serving small allocation requests. **Slab bitmaps** are located in slab headers, with each bit denoting the state (allocated or free) of a slab block. **Heap files** are files that reside on the DAX file system in persistent memory and are mapped as a persistent heap.

Thread-local cache (tcache) tracks addresses of a distinct list of free blocks assembled from local free requests, which may come from multiple slabs. When an allocator receives a request, it searches the tcache first to serve the request. When a block is freed, it goes to the tcache of the thread that frees it, not the one where it was allocated from previously. We use the LIFO algorithm to manage the tcache. When tcache is empty, it is refilled with block addresses from slabs.

Write-ahead logs (WALs) [31, 37] are used to record changes to heap metadata/data when persistent memory allocators use transactions for fail-safe recovery. WAL entries are designed to save essential metadata (e.g., memory addresses and current values).

2.2 Heap Management in Persistent Memory

Small allocations: Slabs are widely used for small allocations (e.g., < 16 KB) to reduce memory fragmentation. We implement a new slab structure for small allocations in persistent memory leveraging the design principles of existing slab structures (i.e., those in jemalloc [17] and nvm_malloc [37]). Specifically, each slab has a persistent header and a volatile header (called *vslab*). The persistent header stores the metadata that is necessary for recovery, including a bitmap whose bits are sequentially mapped to the following blocks. The volatile *vslab* serves for a fast search of free blocks. It could be rebuilt during failure recovery.

Large allocations: Allocators also need to manage large allocations (e.g., ≥ 16 KB). We use the similar structures in jemalloc as examples. Extents are managed using virtual extent headers (VEHs) in DRAM for efficiently searching, splitting, and coalescing of heap extents. Three lists are used to manage VEHs in NVAlloc. An *activated list* stores the VEHs of allocated extents. A *reclaimed list* stores the VEHs of freed extents with physical persistent memory

[1]The source code for NVAlloc is available at https://github.com/ISCS-ZJU/NVAlloc.

Table 1: Workload configuration in Fragbench.

Workload	Before	Delete	After
W1	Fixed 100 B	90%	Fixed 130 B
W2	Uniform 100-150 B	0%	Uniform 200-250 B
W3	Uniform 100-150 B	90%	Uniform 200-250 B
W4	Uniform 100-200 B	50%	Uniform 1000-2000 B

being mapped to virtual addresses. And a *retained list* stores the VEHs of free extents that only have virtual addresses allocated and their physical memory has been unmapped in the process address space.

Upon serving a large allocation, allocators search the reclaimed list and retained list using the first-fit algorithm. If a block is found, its VEH will be moved to the activated list. If none is found, a new VEH is created and added to the activated list. When an extent is freed, it is returned to the reclaimed list. NVAlloc uses a decay-based approach to manage free extents in the reclaimed list and retained list [17]. It uses a *smootherstep* function to calculate the maximum amount of memory TH_{max} that can be used by the lists. If the memory usage of the reclaimed list is higher than TH_{max}, its extents will be moved from the reclaimed list to the retained list. Similarly, if the memory usage of the retained list is higher than its threshold, its extents will be moved to OS. When a VEH is removed from the retained list, its corresponding extent is unmapped in the process address space and its header and extent are freed in persistent memory. A similar approach has been used in the existing work (e.g., jemalloc). We use the same parameters of the *smootherstep* function and time intervals (i.e., 50 ms) as those set in jemalloc.

3 MOTIVATION

We experimentally investigate the performance issue and memory fragmentation induced by the poor heap metadata management in the existing allocators. We use different applications to generate workloads exposing various internal issues.

3.1 Allocator-Induced Cache Line Reflushes

A cache line reflush occurs when repeatedly flushing the same CPU cache line to persistent memory for heap data/metadata persistence. The latency of cache line reflushes is determined by the reflush distance between two accesses to the same cache line. When accessing persistent memory becomes a performance bottleneck in allocators, we can quantify the reflush distance as the number of accesses to unique cache lines. For example, given a sequence of cache lines (A, B, C, D, A) that are flushed consecutively, the reflush distance of cache line A is 3. Our experiment shows that the latency of cache line reflushes is decreased from 800 ns to 500 ns when reflush distance is increased from 0 to 3. In this paper, we assume a cache line reflush occurs when its reflush distance is smaller than 4. Otherwise, a regular flush occurs. We choose 4 as the representative reflush distance because we observe that most cache line reflush distance is smaller than 4 and a larger distance leads to a smaller performance degradation. The average latency of cache line reflushes is 3x and 7x higher than random and sequential writes in persistent memory [7], respectively.

To study the number of allocator-induced cache line reflushes, we run four well-known benchmarks including *Threadtest*, *Prod-con*, *Shbench*, and *Larson*. The details of the experimental setting are presented in Section 6. The percentage of both cache line reflushes and regular flushes are shown in Figure 1(a). We observe that the number of cache line reflushes accounts for up to 99.7%, 94.4%, and 98.8% of the total number of flush operations when running PMDK, nvm_malloc, and PAllocator respectively. This is because they consecutively update the small metadata objects in slab headers or WALs or both to maintain strong consistency. These cache line reflushes slow down the allocation and deallocation operations.

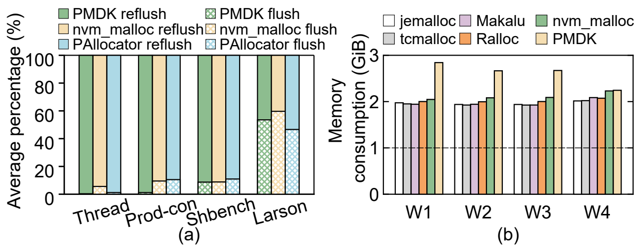


Figure 1: (a) Ratio of cache line reflush; (b) peak memory consumption.

3.2 Fragmentation Caused by Static Slab Segregation

For allocating small objects, slabs are widely used in the existing allocators including volatile memory allocators (e.g., jemalloc-5.2.1 [17] and tcmalloc-2.9.1 [18]) and persistent memory allocators (e.g., Makalu [3], Ralloc [4], nvm_malloc [37], and PMDK-1.11 [11]). Slabs are segregated based on size classes. The size classes are determined when a slab is initialized and cannot be changed at runtime. However, the request size of memory allocation is changing over the execution of server applications [35, 38]. We run the fragmentation benchmark [35] (which we refer to as Fragbench) to study the memory usage of popular allocators. Fragbench has three execution phases: *Before*, *Delete*, and *After*. In the *Before* and *After* phases, Fragbench allocates 5 GB of memory using objects from a pre-defined size distribution and randomly deletes existing objects to keep the amount of live data from exceeding 1 GB. In the *Delete* phase, Fragbench deletes objects randomly. The three phases are executed in order. We change the object size distribution and the ratio of deleted objects in four representative workloads[2] (W1-W4 as shown in Table 1) derived from the benchmark to cover a wide range of characteristics of real-world applications. Similar workloads have been used in the prior research (i.e., RAMCloud [35], PAllocator [31], and log-structured NVMM [20]).

The peak memory consumption is presented in Figure 1(b). To manage the 1 GB live heap data, existing allocators require memory usage of up to 2.8 GB. This result indicates the persistent memory is severely under-utilized. The reason is static slab segregation used in the existing allocators responds to the change of request sizes by allocating more slabs in other size classes [21]. It cannot use the free space in the existing slabs of different size classes. This is because the allocators cannot change a slab's size class at runtime until it is completely free. The memory fragmentation caused by

[2] Although there are eight workloads in the original Fragbench, we only choose the four workloads because other workloads show similar issues and the space is limited.

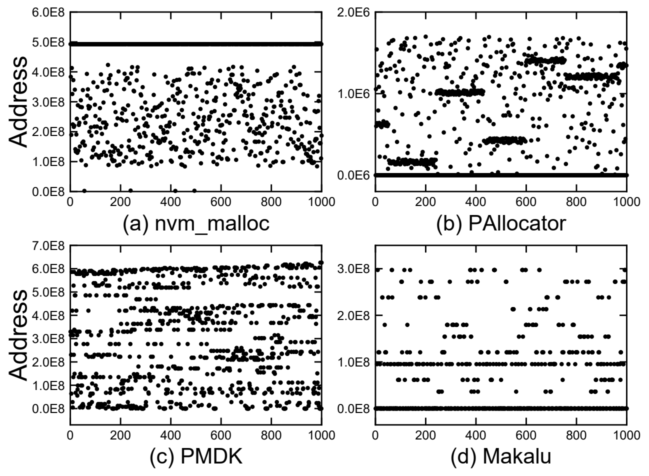


Figure 2: Small random writes in persistent memory. The X-axis denotes the number of flushes.

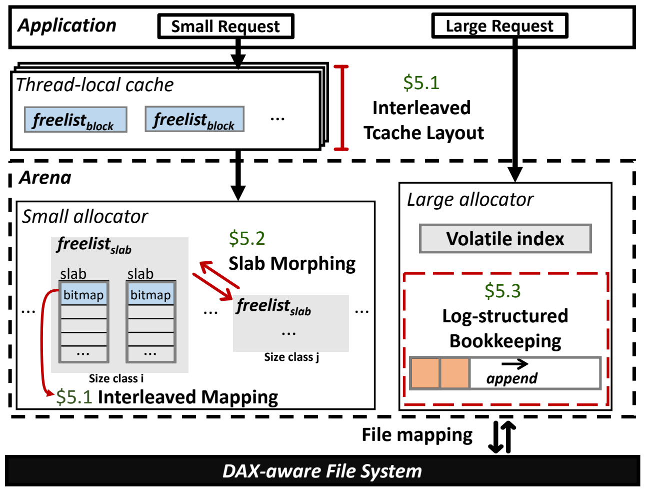


Figure 3: Overview of NVAlloc.

static slab segregation in persistent memory has a larger impact than in volatile memory because the memory fragments cannot be eliminated by restarting.

3.3 Allocator-Induced Small Random Access

For large allocations, most modern allocators (e.g., PMDK and Makalu) store bookkeeping metadata in the header space of a large memory region (e.g., 4 MB). The bookkeeping metadata tracks all extents in the region. The header space is typically placed in a dedicated location separated from heap data space. This layout avoids the header space being modified by users mistakenly. Updating the metadata (e.g., bitmaps and logs) in the header space requires small writes to persistent memory. To study its access pattern, we profile the memory addresses of the first 1000 flush operations of metadata when running the DBMStest benchmark [16] using 4 allocators including nvm_malloc, PAllocator, PMDK, and Makalu. We show the results in Figure 2. We observe that for managing the bookkeeping metadata allocators issue a large number of small random writes to persistent memory and the request addresses are distributed in the whole heap space. The reason is that to serve a large request allocators typically use allocation algorithms (i.e., best-fit, first-fit, or their variants) to find an extent of the most suitable size. After that, they in-place update the bookkeeping metadata in the header space. After a sequence of allocations and deallocations, the best extent candidate can be located in any memory region in the heap space, leading to small random accesses for updating its bookkeeping metadata.

4 NVALLOC

In this section, we present the programming model of NVAlloc and describe the design of its major components: small allocator and large allocator. The NVAlloc software is developed with three optimizations including interleaved mapping which reduces cache line reflushes, slab morphing which alleviates segregation-induced fragmentation, and log-structured bookkeeping which improves the write locality. We illustrate all the components of NVAlloc and where each optimization is applied in Figure 3.

4.1 Programming Model

We use `nvalloc_init()` to create a new NVAlloc instance and `nvalloc_exit()` to safely exit. To avoid the memory leak, we adopt the `nvalloc_malloc_to()` and `nvalloc_free_from()` API used in other allocators [11, 31, 37] to atomically allocate and free objects on persistent memory, respectively. Function `nvalloc_malloc_to()` allocates a block or an extent according to user-specified *size* in the persistent heap and attaches it persistently at a user-specified *address*. We use an offset-based pointer representation to allow persistent structures to be mapped at different virtual addresses after failure recovery. The same technique has been used in previous projects [4, 6, 9]. The `nvalloc_free_from()` returns a block or an extent specified by *address* to the persistent memory heap.

Currently, we implement two variants: NVAlloc-LOG supporting log-based transactional model and NVAlloc-GC supporting GC-based model (see Section 7). As future work, we wish to implement another variant using internal collection for failure recovery. We apply the three novel optimization techniques to various components in these allocators. As an example, we apply interleaved mapping to its WAL, bookkeeping log, bitmaps, and tcache in NVAlloc-LOG. Table 2 shows the details.

For small allocations, NVAlloc-LOG writes all metadata updates in WALs and flushes them to persistent memory to enforce consistency. All memory leaks can be resolved by replaying the WALs. In NVAlloc-GC, no metadata or WALs flushing is used for small allocations to achieve the best runtime performance. However, it needs to execute the post-crash GC during recovery to rebuild heap metadata and check memory leaks, which blocks the normal execution of applications [3]. For large allocations, NVAlloc-GC has the same code path as NVAlloc-LOG.

4.2 Small Allocator

For small allocation (<16 KB), NVAlloc implements *arena* and *tcache* to reduce the thread contention. Each CPU core owns an arena, while each thread owns a tcache. Each thread will be assigned to an arena which has the least number of assigned threads. An arena

Table 2: Techniques used in the two variants of NVAlloc (IM means interleaved mapping).

Allocator	Small allocation	Large allocation
NVAlloc-LOG	IM(WAL,bitmaps,tcache) Slab morphing	IM(WAL,bookkeeping log) Log-structured bookkeeping
NVAlloc-GC	Slab morphing	IM(WAL,bookkeeping log) Log-structured bookkeeping

maintains one freelist of slabs ($freelist_{slab}$) for every size class. The slabs in the freelists are partially full. A tcache maintains one freelist of blocks per size class ($freelist_{block}$). Each block in the freelist is ready to serve an allocation.

When a small block of a certain size is requested, the working thread gets its size class, and then tries to get a block from the corresponding $freelist_{block}$ in tcache. If $freelist_{block}$ is empty, the working thread will fill it until full using slabs from their corresponding $freelist_{slab}$ in the arena. Thread synchronization is required here because multiple threads may be attached to the same arena. If there is no slab in $freelist_{slab}$, it will first use slab morphing (Section 5.2) to find blocks of other size classes to fill tcache. When no blocks can be found using slab morphing, it will require a new slab by executing a large allocation. Once $freelist_{block}$ is filled, users can retrieve a block from tcache immediately.

When a user releases a small block, the working thread will first use an R-tree to find its size class. Then, it is returned to its corresponding tcache. When $freelist_{block}$ is full, the working thread will return the small block to its slab directly, bypassing tcache.

NVAlloc uses interleaved mapping of slab bitmaps and interleaved layout of tcache (Section 5.1) to avoid cache line reflushes when small heap metadata accesses are required.

4.3 Large Allocator

The large allocator in NVAlloc is responsible for allocating slabs and extents that are ranging from 16 KB to 2 MB. For objects larger than 2 MB, NVAlloc calls `mmap()` to allocate a given size extent. The architecture of the large allocator is shown in Figure 7. When `nvalloc_malloc_to()` is called, it first searches the reclaimed list using the best-fit algorithm. If no extent is found, the search is repeated using the retained list. If an extent is found, its virtual extent header (VEH) is moved to the activated list. The extent may need to be split if the existing extent is larger than the request size. To facilitate the extent splitting and coalescing, NVAlloc maintains an R-tree in DRAM to help search the neighboring extents. Each item in the R-tree is a key-value pair, whose key is the start/end address of an extent and value is a pointer to its corresponding VEH. If no extents are available in either the reclaimed list or the retained list, NVAlloc calls `mmap()` to allocate a new extent of 4 MB, which is split into two parts. NVAlloc returns the first part to user and adds it to the activated list. The second part is added to the reclaimed list. Finally, for each part, NVAlloc adds an item to the R-tree pointing to the VEH for the part.

When `nvalloc_free_from()` is called to free a large memory region, NVAlloc searches for its VEH in the R-tree using its memory address. The VEH is moved from the activated list to the reclaimed list. NVAlloc uses a decay-based approach to manage VEHs in the

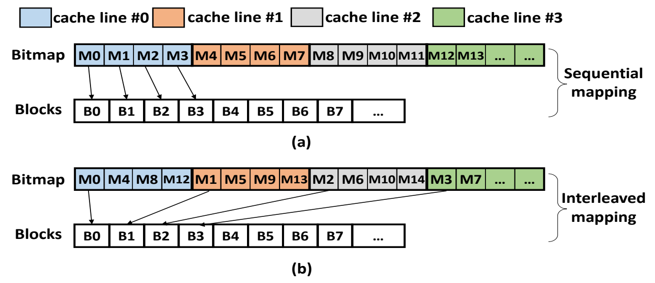

cache line #0
cache line #1
cache line #2
cache line #3
Bitmap
M0 M1 M2 M3 M4 M5 M6 M7 M8 M9 M10 M11 M12 M13 ... ...
Sequential mapping
Blocks
B0 B1 B2 B3 B4 B5 B6 B7 ...
(a)
Bitmap
M0 M4 M8 M12 M1 M5 M9 M13 M2 M6 M10 M14 M3 M7 ... ...
Interleaved mapping
Blocks
B0 B1 B2 B3 B4 B5 B6 B7 ...
(b)

Figure 4: Mapping bitmap to data blocks.

reclaimed list and retained list (see Section 2.2). For failure recovery, when a VEH is created or updated, its essential metadata is added to the persistent bookkeeping log. The operations of the persistent bookkeeping log are described in Section 5.3.

4.4 Recovery

After the recovery process, allocators must ensure that there is no persistent memory leak, and the metadata of the allocators is consistent. Then, the application can conduct normal allocation and deallocation in the persistent heap again.

We use a per-arena flag to mark the states of an arena including *running*, *normal shutdown*, and *recovery*. We change the state to normal shutdown when `nvalloc_exit()` is completed. If the recovery process finds the flag is *running* or *recovery*, it indicates a failure has occurred during running or recovery. In this case, we need to do an additional sanity check to ensure consistency.

For a normal shutdown recovery, we first recreate an arena for each CPU core, and then open and map their respective heap files and log files. After that, for each arena, we perform a slow GC on the persistent bookkeeping log to clean up its tombstone entries (see Section 5.3). Then, we scan and process every log entry. Specifically, for each log entry, we first check its type to determine whether its corresponding extent is a slab. For the slab, we reconstruct its volatile *vslab* based on the metadata in the slab header and add it to the $freelist_{slab}$. Next, we read its $flag$ field to identify whether a slab was morphing when a normal shutdown happened. If it is a $slab_{in}$ (see Section 5.2), we will reconstruct its cnt_{block} and cnt_{slab} additionally. For normal extents, we reconstruct their VEHs and add them to the activated list. We also treat the space gaps between active extents as free extents and insert their VEHs to the reclaimed list in DRAM.

Upon a failure recovery, we first conduct the normal-shutdown recovery. Then, we additionally use different methods to do a memory sanity check to resolve possible memory leaks according to the consistency model of allocators. For NVAlloc-LOG, we replay WALs as in nvm_malloc. For NVAlloc-GC, we conduct conservative garbage collection [3] as in Makalu. As for slabs, we will read the $flag$ field in the slab header to identify whether there is a failure during slab morphing. If a failure is detected, we undo all the operations of metadata transformation.

5 OPTIMIZATION OF METADATA MANAGEMENT

In this section, we introduce three optimizations which address the metadata management issues in persistent memory allocators.

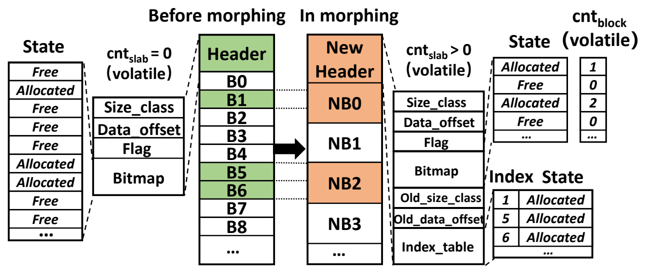


Figure 5: Illustration of slab morphing.

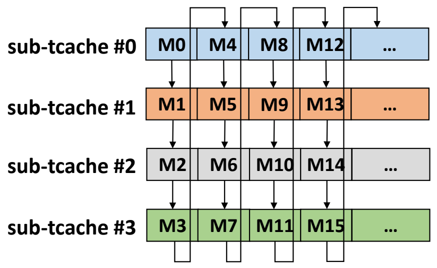


Figure 6: Interleaved tcache layout.

5.1 Interleaved Mapping

Using the slab structure, contiguous small allocations from the same slab need to update consecutive bits in slab bitmaps. Because these bits are likely stored in one CPU cache line, it may cause allocator-induced repeated cache line flushes, leading to longer request latency. A naive approach is allocating blocks at random offsets. Thus, multiple cache lines may be accessed in a random order avoiding reflushing the same cache line. However, this approach compromises the spatial locality of blocks in a persistent heap. Another approach used in the previous work [3, 4] is managing free blocks in a slab using a linked list rather than bitmaps. Each free block has an embedded link pointer. There are three issues with this design. First, placing a header right before the allocated data blocks is prone to metadata corruption from memory corruption bugs [15]. Second, the size of link pointers is much smaller than the size of a cache line. When the link pointers and their corresponding data blocks are stored in the same cache line, allocator-induced reflushes are still possible. Third, blocks in tcache may still be mapped to the same cache line. Therefore, none of the existing work completely solves the problem.

We design a two-level interleaving scheme to produce a metadata layout that eliminates cache line reflushes while maintaining the spatial locality of blocks.

Interleaved mapping of slab bitmaps. Assume we have a bitmap, which has N bits in total. We divide the bitmap into bit stripes, each of which is mapped to a cache line. The stripe size d is the total number of bits in a stripe and is capped by the cache line size. We then map consecutive blocks to bits in different stripes in an interleaved manner. We use Figure 4 for illustration. In this example, we assume the number of bit stripes is 4. In the baseline, bits are sequentially mapped to the data blocks. For example, bits $M0$, $M1$, and $M2$ are mapped to data blocks $B0$, $B1$, and $B2$, respectively. As allocators need to persist the bitmap upon each allocation for crash consistency, contiguous allocations of $B0$, $B1$, and $B2$ result in reflushing the same cache line storing bits $M0$, $M1$, and $M2$. In the interleaved mapping, $M0$, $M1$, and $M2$ are placed in different bit stripes and cache lines. Because $M0$, $M1$, and $M2$ are respectively stored in cache line #0, #1, and #2, there will be no cache line reflush when $B0$, $B1$, and $B2$ are allocated in the slab.

Interleaved layout of tcache. When tcache is used, the order of block allocation is determined by the LIFO algorithm managing tcache. Therefore, it is still possible to have cache line reflushes of contiguous allocations if the bits of blocks selected by tcache are mapped to the same cache line. To avoid cache line reflush issue, we design a new interleaved tcache layout (shown in Figure 6). Specifically, we divide a tcache into multiple sub-tcaches. The number of sub-tcaches is determined by the number of bit stripes. Each sub-tcache caches addresses of blocks whose corresponding bits are mapped to the same cache line. We maintain a cursor to indicate which sub-tcache is used for current allocation. The cursor points to the next sub-tcache after one allocation, which ensures that sub-tcaches mapped to different cache lines are used to serve contiguous allocations. For example, assume that tcache is filled with blocks corresponding to bits $M0$ to $M15$ in Figure 6. Because tcache selects the blocks alternatively from the 4 sub-tcaches for serving contiguous small allocations, we can guarantee that tcache does not select bits mapped to the same cache line. Consequently, cache line reflushes are effectively eliminated.

5.2 Slab Morphing

The existing allocators use static slab segregation to manage slabs, leading to memory fragmentation. We design a new technique, named slab morphing, to address this issue. The idea is that when memory usage of a slab is low, NVAlloc allows it to be transformed to a slab of another size class. During the transformation, the slab may store two types of data blocks of different sizes. We need to address two challenges in the design of slab morphing. (1) The scheme needs to guarantee the correctness of indexing two types of blocks belonging to different size classes. (2) We need to minimize the overhead of managing these blocks.

Block allocation using slab morphing. We manage all the slabs using an LRU list. The slab that is least recently accessed is placed at the head of the list. Slab morphing is only enabled when a small object request comes but existing slabs of the request size class have no space. NVAlloc will choose a slab for morphing and transforming its metadata.

Selecting a slab candidate for morphing. NVAlloc scans the LRU list from head to tail and chooses a slab for morphing when its $Ratio_{occupy}$ is lower than a threshold of space utilization (SU), where $Ratio_{occupy}$ is defined as the ratio of the number of allocated blocks to the number of total blocks in the slab. We set SU as 20% in its current design (see Section 6.5). Because slab morphing needs to change the format of slab headers, a slab will not be selected if the new header space is overlapped with block spaces having live data.

Transforming slab metadata. Then, NVAlloc needs to reset the metadata of the chosen slab. For the convenience of our discussion, we call the slab before, in, and after morphing $slab_{before}$,

$slab_{in}$, and $slab_{after}$ respectively; we refer to the blocks allocated in $slab_{before}$ as $block_{before}$. $Slab_{before}$ and $slab_{after}$ are regular slabs whose headers consist of a *size_class* field, a *data_offset* field (the offset of the starting address of the data region relative to the starting address of a slab), and its *bitmap* field. $Slab_{in}$ needs to support indexing blocks of two size classes. Therefore, we add additional metadata to help implement this functionality. Specifically, we add an *old_size_class* field and *old_data_offset* field in the header of $slab_{in}$ to support the index of $block_{before}$. We also add an *index_table* to ensure the recoverability of $block_{before}$. Each $block_{before}$ has an index table entry, which stores its block index in $slab_{before}$ and its allocation state (i.e., allocated or free). The index table has a small memory footprint because (1) each table entry is only 2 B and (2) we only have a limited number of $blocks_{before}$ since we only select a slab for morphing when its slab usage is low. Finally, we add a counter cnt_{slab} in the volatile header *vslab* to denote the number of allocated $block_{before}$ in the slab. If $cnt_{slab} > 0$, the slab is a $slab_{in}$, otherwise it is a regular slab. We also maintain a counter cnt_{block} in the volatile memory for each block in the $slab_{in}$ to denote the number of $block_{before}$ that occupy it. The cnt_{block} is used to indicate whether a new block can be safely released.

We transform metadata in the following steps. Step 1: set the *old_size_class* and *old_data_offset*; Step 2: set the *index_table*; Step 3: set the *size_class*, *data_offset*, and *bitmap* in the new slab header. Because slab transforming involves multiple steps of modification of metadata, we add a *flag* field to indicate the step of transformation to ensure crash consistency. *Flag* is set to 0 for $slab_{in}$ and $slab_{after}$. During the transformation, we atomically increment *flag* by 1 after each step. *Size_class*, *data_offset*, and allocation information in the *bitmap* will be changed after we have a copy of them in *old_size_class*, *old_data_offset*, and *index_table*. We can undo the morphing if a crash happens during the transformation using *flag*, which denotes which step has been completed. After metadata transformation, $slab_{in}$ is removed from the LRU list because it cannot morph again. It is also removed from the slab list of *old_size_class* and inserted into the slab list of *size_class*.

Figure 5 shows an example of transforming a slab of a small size class to a slab of a large size class with the slab morphing technique. Before morphing, $B1$, $B5$, and $B6$ are allocated in $slab_{before}$. During the transformation, cnt_{block} are set for each block. For $NB0$, its cnt_{block} is set to 1 because only $B1$ of $slab_{before}$ is occupied in $NB0$. For $NB2$, its cnt_{block} is set to 2 because both $B5$ and $B6$ are occupied in $NB2$. Note that the slab morphing also supports slab transforming from a large size class to a small size class.

Block release. When a block is released, NVAlloc determines whether it is a block in $slab_{before}$ by querying cnt_{slab} and cnt_{block}. $Block_{before}$ will be directly put back to $slab_{in}$ bypassing tcache with its state set to *free* in *index_table*. When cnt_{slab} becomes 0, $slab_{in}$ is reset to a regular slab $slab_{after}$ and is inserted into the LRU list again.

The slab morphing scheme introduces a small overhead because it is enabled infrequently. For allocation and release of blocks of a new size class, blocks in $slab_{in}$ can be used to fill the tcache as normal blocks without extra overhead. For the release of $blocks_{before}$, NVAlloc needs to modify its state in the index_table and flush it.

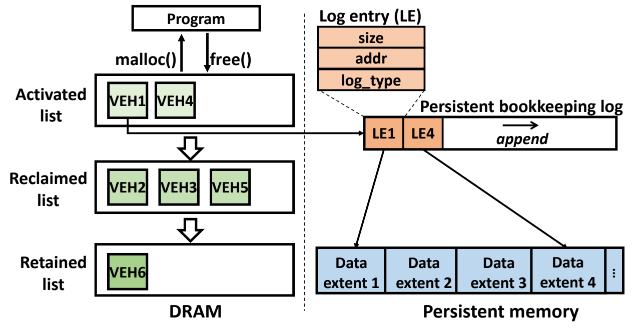


Figure 7: Illustration of log-structured bookkeeping.

These operations have a low cost because $blocks_{before}$ only account for up to 20% of the total blocks as set in our experiments. We quantify its overhead in Section 6.4.

5.3 Log-Structured Bookkeeping

For large allocation and release, NVAlloc uses a virtual extent header (VEH) in DRAM to manage every extent in persistent memory. VEHs are moved between the activated list, the reclaimed list, and the retained list. The essential metadata (e.g., size and address) needs to be updated in their corresponding extent headers in persistent memory. Because of in-place extent header updates, random access to the headers is unavoidable. To eliminate the random accesses induced by large allocations, we design a log-structured bookkeeping scheme as shown in Figure 7. Specifically, when a virtual extent header (e.g., $VEH1$) is updated, its essential metadata is appended to a persistent bookkeeping log. The log is sequentially written and cleaned up when it is full. We trade persistent memory space for better spatial locality.

The overhead of log-structured bookkeeping in allocators is very low for the following reasons. First, the persistent bookkeeping log only stores small essential metadata. Each log entry is only 8 B, consisting of 26 bits for "size", 36 bits for "addr", and 2 bits for "log type". For "addr", we only need 36 bits because (1) only the low-order 48 bits are used in 64-bit address space in Intel x86 processors [31] and (2) our address is 4 KB-aligned, thus the lower 12 bits are not needed in the log entry. This is different from traditional log-structured file systems, whose log entry can be as large as a request size. Consequently, the space overhead of metadata logging is much smaller than data logging in traditional log-structured file systems. Therefore, we can afford to trade more space for a better space locality without incurring the overhead of garbage collection. Second, log entry size is unified in persistent bookkeeping logs, leading to a simplified log management process.

One major challenge is cache line reflushes for writing small log entries. We introduce the layout of persistent bookkeeping logs and how to prevent cache line reflushes in logging and how to reduce GC overhead.

The layout of persistent bookkeeping log. The persistent bookkeeping log has two components in DRAM and persistent memory, respectively. Its layout is shown in Figure 8. At the time of initialization, NVAlloc creates a file of 100 MB in persistent memory to store log entries. A log file is divided into chunks of 1 KB, each

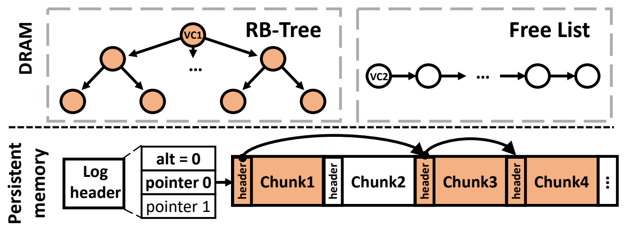


Figure 8: The memory layout of the persistent bookkeeping log. VC denotes vchunk. Chunks in orange and white color denote active and free chunks respectively.

of which can store 128 log entries. The chunks are managed as a linked list. The log file has a log header, which stores two pointers and an *alt* bit. One of the pointers refers to the head of the linked list of active log chunks upon recovery; the other one is only used by GC for building a new linked list. The *alt* bit indicates which one of the two pointers is active. Each chunk has a chunk header, which stores its ID number, an activeness bit, and a pointer to the next active chunk.

To speed up the log operation, each log chunk has a corresponding *volatile chunk, vchunk* in DRAM. It stores a bitmap indicating the valid log entries in the chunk. Besides, NVAlloc uses a red-black tree to manage the vchunks of the allocated chunks. After GC, all the freed chunks are retained in a linked list for fast allocation in the future. When a new log chunk is needed, it is first retrieved from the free list. If the free list is empty, a new chunk is created and appended to the tail of the log file in persistent memory.

Basic log operation. In NVAlloc, the log entry has two different types: normal entry and tombstone entry. When allocating a large block, a normal entry will be created and added in the current chunk. To avoid cache line reflushes, we map consecutive log entries to the chunk in an interleaved manner, similar to the method in Section 5.1. Then the corresponding bit in the bitmap of its vchunk will be set.

Similar to the normal entries, a tombstone entry will be added when freeing a large extent. In addition, the tombstone entry will store the pointer of the normal entry to be deleted and clean its corresponding bit in the bitmap of vchunk for fast garbage collection.

Garbage collection (GC). To control the size of the log file, we need to execute garbage collection (GC) to drop the log entries that are marked as deleted by the tombstones. NVAlloc supports two GC algorithms including *fast GC* and *slow GC* [20], which are designed to make different tradeoffs between the GC overhead and memory efficiency.

The fast GC algorithm scans the bitmap of each vchunk in the red-black tree. If the bitmap of a vchunk is empty, it will be moved to the free list. Because the fast GC algorithm does not need to access persistent memory, its overhead is trivial. NVAlloc executes fast GC most of the time and only switches to execute slow GC when the size of the log file is larger than a memory usage threshold $Usage_{pmem}$. When the slow GC algorithm is executed, a new active chunk list $list_{new}$ will be created to store the active log entries. The slow GC algorithm scans all the log entries in the existing active chunk list $list_{old}$ and checks whether the log entries are alive via bitmap. The log entries that are alive in $list_{old}$ will be copied to chunks in $list_{new}$. The tombstone entries will be removed in the process. When the scanning is completed, NVAlloc marks $list_{new}$ as the current active chunk list by flipping the *alt* bit. Then it recycles all the chunks in $list_{old}$.

6 EVALUATION

6.1 Experimental Setup

Experimental platform. We run the experiments on a Linux server (kernel 5.3.0-050300-generic) with two Intel Xeon Gold 5218R CPUs. Each CPU has 20 physical cores (40 hyperthreads), 64 GB DRAM, and two Intel Optane DIMMs (128 GB per DIMM). Every pair of DIMMs attached to a CPU is mounted with the Ext4-DAX file system and configured in App Direct mode. To avoid the NUMA effects, we use the numactl utility to bind every thread to one core in the first socket. All source codes are compiled with g++7.5 with -O3.

Compared allocators. We compare NVAlloc with state-of-the-art persistent allocators, including PMDK [11], nvm_malloc [37], PAllocator [31], Makalu [3], and Ralloc [4]. Since all of them except PAllocator are open-source, we use their public implementations for tests. We reimplement PAllocator as faithfully as possible according to the description in the paper. We exclude jemalloc [17], Hoard [2], and tcmalloc [18] because they are volatile allocators. To support existing consistency models, we implement two versions of NVAlloc: *NVAlloc-LOG* and *NVAlloc-GC*, which leverage WAL and GC to keep crash consistency and avoid memory leaks, respectively.

For ease of description, we call PMDK and WAL-based allocators (i.e., nvm_malloc, PAllocator, and NVAlloc-LOG) as *strongly consistent allocators*. In contrast, we call GC-based allocators (i.e., Makalu, Ralloc, and NVAlloc-GC) *weakly consistent allocators*.

6.2 Evaluations using Benchmarks

Benchmarks. We use five representative benchmarks, each of which has a unique allocation pattern, in the evaluation.

Threadtest [2] measures multi-threaded performance of an allocator for i iterations of allocations. In every iteration, each thread allocates n objects in size of s and then frees all of them independently. In the experiment, we set $i = 10^4$, $n = 10^5$, and $s = 64\,\text{B}$.

Prod-con [2, 36] simulates a producer-consumer workload for t threads. Each pair of threads produces and consumes n objects, whose total size is s. One thread of each pair allocates objects while the other one frees them. Our experiment sets $n = \frac{2\times10^7}{t}$ and $s = 64\,\text{B}$.

Shbench [29] is a stress test for an allocator. In each iteration, each thread allocates and frees objects of varying sizes from 64 B to 1000 B. The smaller objects are allocated and freed more frequently. We run 10^5 iterations.

Larson [22, 31] simulates a behavior where some objects allocated by one thread are freed by another thread. In each iteration, each thread randomly allocates and frees 10^3 varied-size objects. After 10^4 iterations, each thread creates a new thread that starts with the remaining objects and repeats the same allocation/deallocation procedure. We generate two workloads: *Larson-small* managing small objects (64 B to 256 B) and *Larson-large* managing large objects (32 KB to 512 KB). We run the test for 30 seconds.

DBMStest [16]: it simulates the allocation in a database with TPC-DS benchmark for t threads. In each iteration, each thread

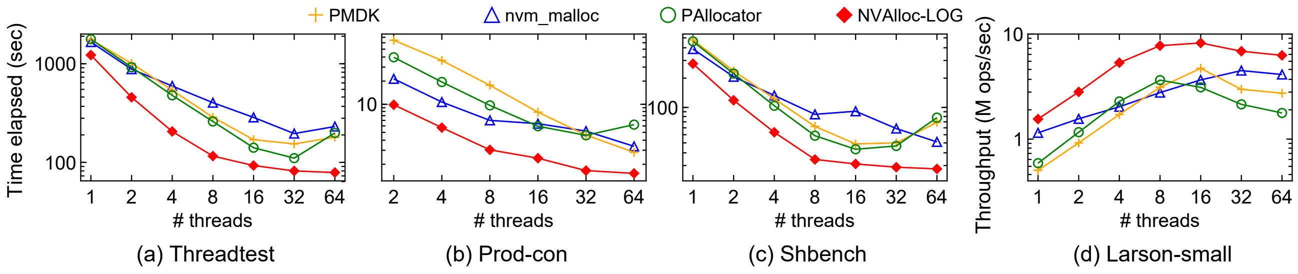


Figure 9: Performance (log10 scaled) of small allocations with strongly consistent allocators.

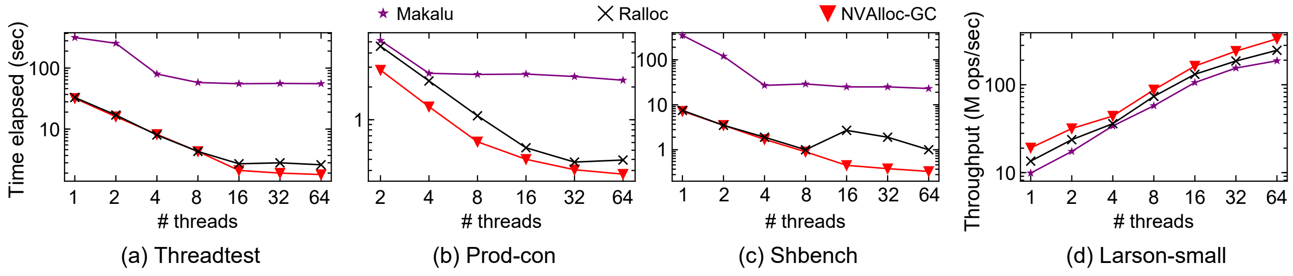


Figure 10: Performance (log10 scaled) of small allocations with weakly consistent allocators.

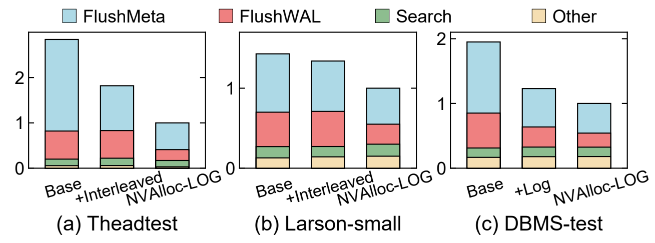


Figure 11: Performance breakdown analysis.

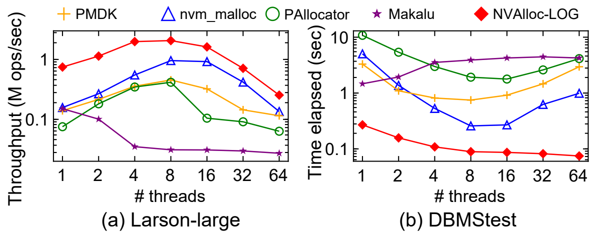


Figure 12: Performance (log10 scaled) of large allocations.

allocates n large objects, whose sizes follow a Poisson distribution between 32 KB to 512 KB, and then randomly deletes 90% of them. We choose $n = \frac{10^4}{t}$ objects. We run 50 iterations for warmup and 50 iterations for evaluation.

Performance of small allocations. We first evaluate the allocator performance for small object allocations with varying numbers of threads on Threadtest, Prod-con, Shbench, and Larson-small. For a fair comparison, we show the results of strongly and weakly consistent allocators in Figure 9 and Figure 10, respectively. Overall, NVAlloc outperforms and scales better than all the counterparts on all benchmarks.

Figure 9 shows that NVAlloc-LOG is up to 6.4x, 3.5x, and 3.9x faster than PMDK, nvm_malloc, and PAllocator, respectively, on the four benchmarks. NVAlloc-LOG outperforms its counterparts because the interleaved mapping reduces the number of cache line reflushes in both metadata updating and WAL updating. To further analyze these results, we use the linux *perf* tools [14] to measure the breakdown of execution time of different benchmarks. We only evaluate Threadtest, Larson-small, and DBMS-test because other benchmarks show similar results and the space is limited. The execution time consists of object searching, splitting, and coalescing of extents in allocation/deallocation (denoted as Search), metadata flushing time (FlushMeta), WAL flushing time (FlushWAL), and other time (Other). We show the results with eight threads in Figure 11. *Base* denotes NVAlloc-LOG without enabling any optimizations described in Section 4. *+Interleaved* denotes the version in which only interleaved tcache layout is enabled. *+Log* denotes the version in which only log-structured bookkeeping is enabled. As Figure 11(a) shows, the FlushMeta and FlushWAL account for 87% of the execution time of *Base* on Threadtest. *+Interleaved* reduces the FlushMeta time and the total execution time of *Base* by 51% and 35%, respectively. NVAlloc-LOG further reduces the total amount of flush time (FlushMeta and FlushWAL) by 48% when the interleaved mapping is additionally used in both slab bitmaps and WALs.

Figure 11(b) shows the breakdown of the execution time on Larson-small. With the interleaved tcache layout, the FlushMeta time in Base-tcache is reduced by 14% and the total time is reduced by 7%. *+Interleaved* achieves the gain because the tcache selects the blocks from each sub-tcache in turn, avoiding cache line reflushes in updating the persistent bitmaps. With interleaved mapping enabled in both slabs and WALs, NVAlloc-LOG achieves a 31% performance gain over Base. Another observation is that NVAlloc-LOG obtains more benefits on Threadtest compared to Larson-small. The reason

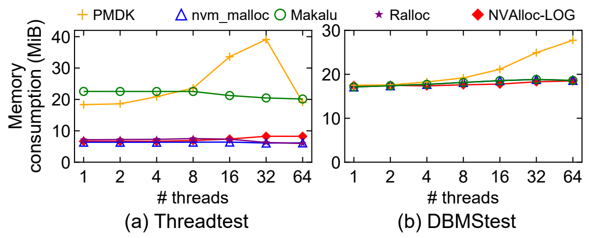


Figure 13: Space consumption.

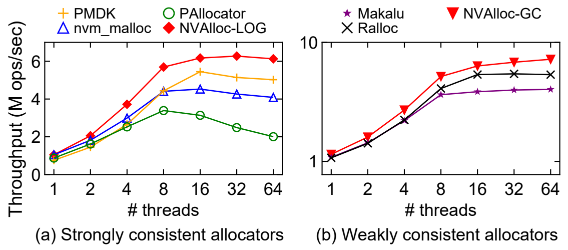


Figure 14: Performance of FPTree.

is that there are more cache line reflushes on Threadtest because of its fixed allocation size.

Figure 10 shows that NVAlloc-GC achieves a maximal speedup of 70x and 6x over Makalu and Ralloc on the four benchmarks. NVAlloc-GC has better performance because Makalu and Ralloc use the embedded linked list to manage free blocks in persistent slabs while NVAlloc-GC uses bitmaps to manage blocks, improving data access locality in persistent memory. NVAlloc-GC also maintains a volatile bitmap copy in DRAM for fast free block indexing and reducing accesses to persistent memory.

Performance of large allocations. Figure 12 shows the performance of large object allocations. Because NVAlloc-GC and NVAlloc-LOG perform the same for large allocations and Ralloc does not work correctly in its open-source implementation for large objects, we exclude them in Figure 12. On Larson-large and DBMStest, NVAlloc-LOG is up to 40x, 18x, 55x, and 57x faster than PMDK, nvm_malloc, PAllocator, and Makalu. NVAlloc-LOG is faster than its counterparts because of using log-structured bookkeeping and the interleaved mapping in WALs.

To show the impact of log-structured bookkeeping and interleaved mapping, we measure the breakdowns of the execution time with DBMS-test. Figure 11(c) shows the results. *+Log* is the version with only log-structured bookkeeping added to *Base*. It reduces the total amount of flush time (FlushMeta and FlushWAL) by 45% because the log-structured bookkeeping provides sequential write pattern to persistent memory. NVAlloc-LOG further reduces the flush time by 26% because the repeated cache line flushes of log entries are eliminated.

Space usage results. Figure 13 shows the memory consumption of different allocators. We only take Threadtest and DBMS as an example for small and large object allocations respectively because other benchmarks exhibit similar results. As the NVAlloc-LOG and NVAlloc-GC show the same space consumption, we only include NVAlloc-LOG. NVAlloc-LOG yields comparable or better space consumption than other allocators on both benchmarks. We exclude RAlloc in Figure 13 (b) because RAlloc does not work correctly for large objects in their open-source implementation.

6.3 Evaluation using FPTree

We also evaluate NVAlloc with a real-world key-value store application, FPTree [32]. It is a persistent concurrent B+tree, which stores the inner nodes in DRAM and the leaf nodes in persistent memory. Each node of FPTree contains 64 children. To support varied-size values, FPTree uses the original value in the leaf node as a pointer to an actual key-value pair. We set the size of original keys and values as 8 B. Since most key-value pairs are small in Facebook [5], we set the size of the actual key-value pair as 128 B. We measure the performance of FPTree with a mixed workload of 50% insertions and 50% delete operations. We warm up the FPTree with 50 M key-value pairs, then execute 50 M operations with a varying number of threads.

Figure 14 shows the throughputs of FPTree using different allocators. With NVAlloc-LOG, FPTree yields up to 1.2x, 1.5x, and 3.1x throughput compared with PMDK, nvm_malloc, and PAllocator, respectively. With NVAlloc-GC, FPTree brings a speedup up to 35.4%. FPTree with NVAlloc yields comparable space consumption over other allocators since the slab morphing technique is not triggered for the given workload.

6.4 Evaluations using Fragbench

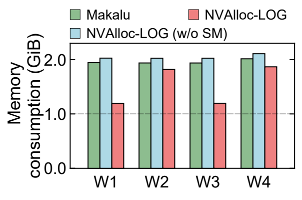
We then evaluate NVAlloc on Fragbench [35] with the four workloads listed in Table 1 in Section 3. Figure 15(a) shows the space consumption of different allocators. We exclude the ones in Figure 1(b) except Makalu to avoid redundant representation. As NVAlloc-LOG and NVAlloc-GC yield the same space consumption, we only include NVAlloc-LOG. For comparison, we also evaluate NVAlloc-LOG without the slab morphing strategy (NVAlloc-LOG w/o SM). The result shows that NVAlloc-LOG achieves the smallest space consumption because of the slab morphing technique.

To verify this, Figure 15(b) shows the space breakdown of NVAlloc-LOG. We divide the slabs into three categories according to their memory utilization: 0-30%, 30-70%, 70-100%. Figure 15(b) shows that, with the slab morphing, NVAlloc-LOG greatly increases the number of slabs with high utilization, compared to the scheme without using slab morphing. Thus, it decreases the overall memory consumption.

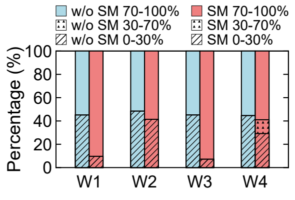
Figure 15(c) and (d) show the performance of NVAlloc. NVAlloc outperforms all other allocators because of using the interleaved mapping technique, as discussed in Section 6.2. We also observe that the slab morphing approach may introduce a performance degradation of 4.5% on average because it needs to flush slab metadata. Despite the slight performance slowdown, the slab morphing reduces memory usage by up to 41.9%.

6.5 Sensitivity Analysis

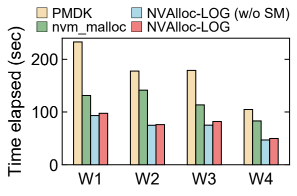
Number of bit stripes. The efficiency of interleaved mapping is related to the number of bit stripes. A larger number of bit stripes decreases the number of reflushes because each bit stripe has fewer bits and thus fewer blocks are mapped to the same cache line. However, it may increase the flushing latency because we may exhaust the XPBuffer [40] in persistent memory when a large number of cache lines flush concurrently. To explore the impact of the number



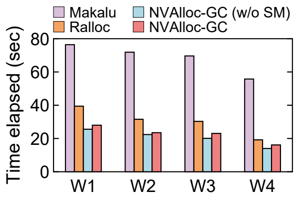
(a) Space consumption



(b) Space breakdown

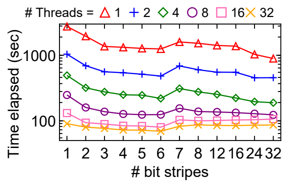


(c) Performance of strongly consistent allocators

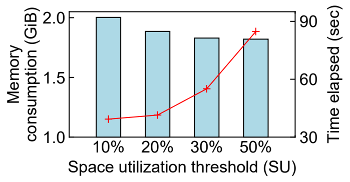


(d) Performance of weakly consistent allocators

Figure 15: Results of Fragbench. SM denotes slab morphing.



(a) Number of bit stripes impact on Threadtest



(b) Morphing parameter impact on Fragbench

Figure 16: Sensitivity Analysis.

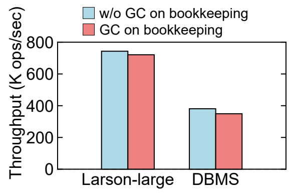


Figure 17: GC overhead.

Allocators	Time
nvm_malloc	324us
PMDK	34ms
NVAlloc-LOG	45ms
Ralloc	552ms
Makalu	911ms
NVAlloc-GC	933ms

Figure 18: Recovery time.

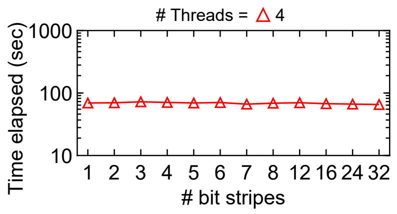


Figure 19: Impact of interleaved mapping on eADR.

of bit stripes, we run NVAlloc-LOG on Threadtest with varying numbers of threads as a study case.

As Figure 16(a) shows, the execution time of NVAlloc-LOG is not linearly decreased as we increase the number of bit stripes. This is because the execution time is determined by both software parameters (i.e., the number of bit stripes and the number of threads) and hardware parameters (i.e., the number of XPBuffer lines in persistent memory and its size). In this paper, we choose the number of bit stripes as 6 because it achieves the best performance for most cases. We leave it as future work to dynamically choose the number of stripes with varying levels of thread concurrency.

Morphing parameter. The slab space utilization threshold (SU) in the morphing technique also impacts the efficiency of NVAlloc. A larger SU allows more slabs to be morphed and thus decreases memory consumption, while a smaller SU decreases the morphing cost and thus improves performance. Figure 16(b) shows the impact of SU on NVAlloc-LOG on the W4 workload. Based on the results, we empirically set SU as 20% to achieve a decent trade-off between memory consumption and allocator performance. While this parameter works well in our initial prototype, using a more sophisticated parameter could be more beneficial. We leave such exploration for future work.

6.6 Overhead Discussion

GC overhead. To evaluate the efficiency of log cleaning on log-structured bookkeeping for large allocations, we run NVAlloc-LOG on Larson-large and DBMStest. Figure 17 shows, with GC, the throughput drops slightly (only 3%) on Larson-large and 8% on DBMS when $Usage_{pmem}$ = 0.2%. The GC overhead is trivial because the log-structured file is light-weight since it only keeps the allocation metadata thus the copying overhead is low.

Recovery. Figure 18 shows the recovery time of open-source allocators. We first create a single linked list with 10 M nodes whose sizes are uniformly distributed between 64 B and 128 B, and then we execute the recovery using a single thread. For strongly consistent allocators, NVAlloc-LOG is slower than PMDK and nvm_malloc because it needs to scan WALs and the log-structure bookkeeping while PMDK only needs to travel the WALs and nvm_malloc defers some metadata reconstruction to the runtime deallocation process. However, NVAlloc-LOG outperforms them in runtime performance. For weakly consistent allocators, NVAlloc-GC performs comparably over Makalu. It is slower than Ralloc because Ralloc only needs to scan part of nodes in the recovery.

6.7 Evaluation on Emulated eADR Platform

eADR (extended ADR) is a new feature supported in the 3rd generation Intel Xeon Scalable Processors, which ensures CPU caches are also in the power fail protected domain [12]. Thus, explicit cache line flushes are not necessary on eADR. Implementing eADR requires higher energy consumption, hardware cost, and system maintenance burden. Given these issues, both ADR and eADR platforms will co-exist in the foreseeable future, as pointed out by Intel [34]. In this section, we evaluate NVAlloc on the eADR platform. Because the eADR is not commercially available, we emulate it by removing flush operations (i.e., *clwb*) on the ADR platform for all evaluated allocators. We only evaluate the strongly consistent allocators because the weakly consistent allocators removed most of the flush operations by performing post-crash GC.

First, we evaluate the impact of interleaved mapping on eADR. We run Threadtest with 4 threads while the number of bit stripes is increased from 1 to 32. As shown in Figure 19, the number of bit stripes has no impact on the performance of NVAlloc-LOG. Because the interleaved mapping increases the cache usage, we disable the interleaved mapping on the emulated eADR platform in the following experiments. For the real eADR platform, we use

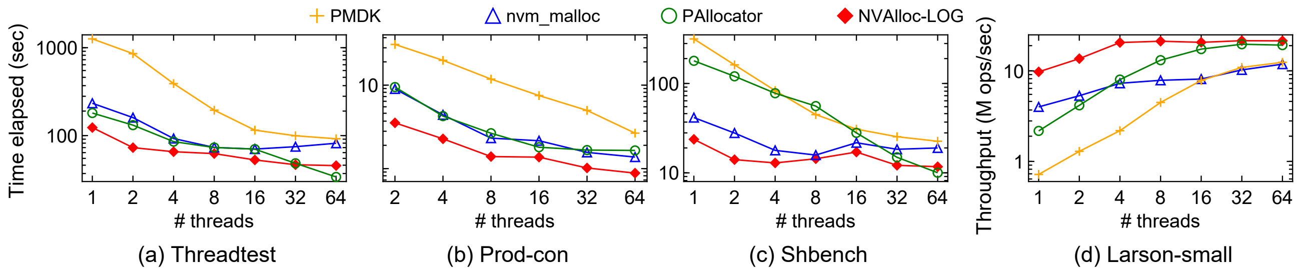


Figure 20: Performance (log10 scaled) of small allocations on the emulated eADR platform.

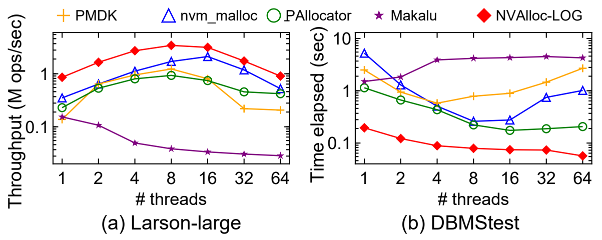


Figure 21: Performance (log10 scaled) of large allocations on the emulated eADR platform.

pmem_has_auto_flush() in PMDK [11] to automatically detect the eADR feature and then disable the interleaved mapping technique.

Second, we study the small allocation performance on eADR. The results in Figure 20 show that NVAlloc-LOG improves the performance of the benchmarks by 240% on average compared to other strongly consistent allocators. The execution time of PAllocator with Threadtest is 27% smaller than that with NVAlloc-LOG when the number of threads is 64. This is because PAllocator uses dedicated small allocators for each thread. It achieves better scalability for thread-local allocations but leads to worse performance of frequent cross-thread operations in Prod-con and Larson-small.

Third, Figure 21 shows the performance of NVAlloc-LOG for large allocations. We can observe that it has an 11x performance improvement on average with Larson-large and DBMStest. This is because our design of VEH and log-structured bookkeeping reduces the total number of persistent memory access and improves the write locality for eADR.

7 RELATED WORK

Log-based allocators. Persistent memory allocators supporting transactional models record changes of memory addresses and heap metadata in logs. After replaying the logs, allocators can rebuild their heap metadata after a crash. For example, nvm_malloc [37] divides its heap metadata into volatile and non-volatile parts to reduce data accesses on persistent memory. Its small writes to its bitmaps and logs may lead to cache line reflushes. PAllocator [31] serves small block allocation using segregated-fit strategy and large block allocation using index trees. It also suffers from the cache line reflush issue because of accessing 2-B block metadata in page headers and micro-logs. Poseidon [15] is the first persistent memory allocator enforcing page-based protections. It also uses bitmaps and logs for heap metadata management.

GC-based allocators. To avoid the overhead of writing logs and flushing metadata, recent allocators [3, 4, 28] use garbage collection (GC) to rebuild heap metadata post crash by traversing the persistent heap from persistent root pointers. Makalu [3] is the first allocator that uses offline GC to relax heap metadata persistence constraints online, resulting in faster small-block allocations. Ralloc [4] turns transient lock-free allocator LRalloc into a persistent allocator. Same as Makalu, Ralloc uses post-crash GC to avoid cache line reflushes. DCMM [28] eliminates the long heap metadata recovery time by simply allocating new blocks appending to the existing heap area and employing background recovery threads running in parallel.

Allocators using internal collection. The allocator in PMDK provides non-transactional atomic allocations [11]. Using PMDK's interface (e.g., `POBJ_FIRST()` and `POBJ_NEXT()`), users will never lose a reference to an object in persistent memory. Therefore, the allocators using PMDK's internal collection do not need to maintain write-ahead logs.

The approaches proposed in NVAlloc can be used to implement log-based, GC-based, or internal-collection-based persistence models. In any of these models, we can eliminate the allocator-induced cache line reflushes and random writes to persistent memory, compared to the existing allocators. Besides, because we use slab morphing, NVAlloc no longer has the memory fragmentation issue caused by static slab segregation.

8 CONCLUSION

In the paper, we design a novel allocator, named NVAlloc, to allocate/deallocate memory objects in persistent memory. NVAlloc leverages interleaved metadata mapping, log-structured bookkeeping, and slab morphing techniques to eliminate the allocator-induced cache line reflushes, small random writes, and memory fragmentation issues. Our experimental results demonstrate that NVAlloc can significantly improve allocator performance and space utilization. As persistent memory becomes more and more popular, we hope the various optimization techniques in NVAlloc will inspire the future generation of persistent memory systems.

ACKNOWLEDGMENTS

We sincerely thank our shepherd Angela Demke Brown and the anonymous reviewers for their constructive suggestions. This work was supported in part by the National Key Research and Development Program of China No.2021ZD0110700, the National Science Foundation of China No. 62172361, the Zhejiang Lab Research Project No. 2020KC0AC01, the US National Science Foundation under CNS 1906541, and the Alibaba Innovative Research Project.

REFERENCES

[1] Joy Arulraj, Andrew Pavlo, and Subramanya R. Dulloor. 2015. Let's Talk About Storage & Recovery Methods for Non-Volatile Memory Database Systems. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD)*. Association for Computing Machinery, 707–722.

[2] Emery D Berger, Kathryn S McKinley, Robert D Blumofe, and Paul R Wilson. 2000. Hoard: A Scalable Memory Allocator for Multithreaded Applications. *ACM Sigplan Notices* 35, 11 (2000), 117–128.

[3] Kumud Bhandari, Dhruva R Chakrabarti, and Hans-J Boehm. 2016. Makalu: Fast Recoverable Allocation of Non-Volatile Memory. *ACM SIGPLAN Notices* 51, 10 (2016), 677–694.

[4] 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)*. 60–73.

[5] Zhichao Cao, Siying Dong, Sagar Vemuri, and David HC Du. 2020. Characterizing, Modeling, and Benchmarking RocksDB Key-Value Workloads at Facebook. In *18th USENIX Conference on File and Storage Technologies (FAST)*. 209–223.

[6] Guoyang Chen, Lei Zhang, Richa Budhiraja, Xipeng Shen, and Youfeng Wu. 2017. Efficient Support of Position Independence on Non-Volatile Memory. In *Proceedings of the 50th Annual IEEE/ACM International Symposium Fon Microarchitecture (MICRO)*. 191–203.

[7] Youmin Chen, Youyou Lu, Fan Yang, Qing Wang, Yang Wang, and Jiwu Shu. 2020. FlatStore: An Efficient Log-Structured Key-Value Storage Engine for Persistent Memory. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 1077–1091.

[8] Zhangyu Chen, Yu Huang, Bo Ding, and Pengfei Zuo. 2020. Lock-Free Concurrent Level Hashing for Persistent Memory. In *Proceedings of the 2020 USENIX Annual Technical Conference (ATC)*. 799–812.

[9] Joel Coburn, Adrian M Caulfield, Ameen Akel, Laura M Grupp, Rajesh K Gupta, Ranjit Jhala, and Steven Swanson. 2011. NV-Heaps: Making Persistent Objects Fast and Safe with Next-Generation, Non-Volatile Memories. *ACM SIGARCH Computer Architecture News* 39, 1 (2011), 105–118.

[10] Intel Corporation. 2018. Redis. https://github.com/pmem/redis/tree/3.2-nvml/.

[11] Intel Corporation. 2020. Persistent Memory Development Kit. http://pmem.io/.

[12] Intel Corporation. 2021. eADR: New Opportunities for Persistent Memory Applications. https://www.intel.com/content/www/us/en/developer/articles/technical/eadr-new-opportunities-for-persistent-memory-applications.html

[13] Andreia Correia, Pascal Felber, and Pedro Ramalhete. 2018. Romulus: Efficient Algorithms for Persistent Transactional Memory. In *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures(SPAA)*. 271–282.

[14] Arnaldo Carvalho De Melo. 2010. The new linux perf tools. In *Slides from Linux Kongress*, Vol. 18. 1–42.

[15] Anthony Demeri, Wook-Hee Kim, R Madhava Krishnan, Jaeho Kim, Mohannad Ismail, and Changwoo Min. 2020. Poseidon: Safe, Fast and Scalable Persistent Memory Allocator. In *Proceedings of the 21st International Middleware Conference (Middleware)*. 207–220.

[16] Dominik Durner, Viktor Leis, and Thomas Neumann. 2019. On the Impact of Memory Allocation on High-Performance Query Processing. In *Proceedings of the 15th International Workshop on Data Management on New Hardware (DaMoN)*. 1–3.

[17] Jason Evans. 2021. jemalloc. https://github.com/jemalloc/jemalloc/.

[18] Inc Google. 2021. tcmalloc. https://github.com/google/tcmalloc.

[19] Jinyu Gu, Qianqian Yu, Xiayang Wang, Zhaoguo Wang, Binyu Zang, Haibing Guan, and Haibo Chen. 2019. Pisces: A Scalable and Efficient Persistent Transactional Memory. In *2019 USENIX Annual Technical Conference (ATC)*. USENIX Association, 913–928.

[20] Qingda Hu, Jinglei Ren, Anirudh Badam, Jiwu Shu, and Thomas Moscibroda. 2017. Log-Structured Non-Volatile Main Memory. In *Proceedings of the 2017 USENIX Annual Technical Conference (ATC)*. 703–717.

[21] Mark S Johnstone and Paul R Wilson. 1998. The Memory Fragmentation Problem: Solved? *ACM Sigplan Notices* 34, 3 (1998), 26–36.

[22] Per-Åke Larson and Murali Krishnan. 1998. Memory Allocation for Long-Running Server Applications. In *Proceedings of the 1st International Symposium on Memory Management (ISMM)*. 176–185.

[23] Se Kwon Lee, K Hyun Lim, Hyunsub Song, Beomseok Nam, and Sam H Noh. 2017. WORT: Write Optimal Radix Tree for Persistent Memory Storage Systems. In *Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST)*. 257–270.

[24] Se Kwon Lee, Jayashree Mohan, Sanidhya Kashyap, Taesoo Kim, and Vijay Chidambaram. 2019. Recipe: Converting Concurrent DRAM Indexes to Persistent-Memory Indexes. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*. 462–477.

[25] Lenovo. 2018. Memcached-PMEM. https://github.com/lenovo/memcached-pmem/.

[26] Jihang Liu, Shimin Chen, and Lujun Wang. 2020. LB+ Trees: Optimizing Persistent Index Performance on 3DXPoint Memory. *Proceedings of the VLDB Endowment* 13, 7 (2020), 1078–1090.

[27] Baotong Lu, Xiangpeng Hao, Tianzheng Wang, and Eric Lo. 2020. Dash: Scalable Hashing on Persistent Memory. *Proc. VLDB Endow.* 13, 8 (2020), 1147–1161.

[28] Shaonan Ma, Kang Chen, Shimin Chen, Mengxing Liu, Jianglang Zhu, Hongbo Kang, and Yongwei Wu. 2021. ROART: Range-Query Optimized Persistent ART. In *Proceedings of the 19th USENIX Conference on File and Storage Technologies (FAST)*. 1–16.

[29] Inc MicroQuill. 2014. shbench. http://www.microquill.com/.

[30] Iulian Moraru, David G Andersen, Michael Kaminsky, Niraj Tolia, Parthasarathy Ranganathan, and Nathan Binkert. 2013. Consistent, Durable, and Safe Memory Management for Byte-Addressable Non-Volatile Main Memory. In *Proceedings of the First ACM SIGOPS Conference on Timely Results in Operating Systems (TRIOS)*. 1–17.

[31] Ismail Oukid, Daniel Booss, Adrien Lespinasse, Wolfgang Lehner, Thomas Willhalm, and Grégoire Gomes. 2017. Memory Management Techniques for Large-Scale Persistent-Main-Memory Systems. *Proceedings of the VLDB Endowment* 10, 11 (2017), 1166–1177.

[32] Ismail Oukid, Johan Lasperas, Anisoara Nica, Thomas Willhalm, and Wolfgang Lehner. 2016. FPTree: A Hybrid SCM-DRAM Persistent and Concurrent B-tree for Storage Class Memory. In *Proceedings of the 2016 International Conference on Management of Data (SIGMOD)*. 371–386.

[33] Bobby Powers, David Tench, Emery D Berger, and Andrew McGregor. 2019. Mesh: Compacting Memory Management for C/C++ Applications. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 333–346.

[34] Andy Rudoff. 2020. Persistent Memory Programming without All That Cache Flushing. *SDC* (2020).

[35] Stephen M Rumble, Ankita Kejriwal, and John Ousterhout. 2014. Log-Structured Memory for DRAM-Based Storage. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST)*. 1–16.

[36] Scott Schneider, Christos D Antonopoulos, and Dimitrios S Nikolopoulos. 2006. Scalable Locality-Conscious Multithreaded Memory Allocation. In *Proceedings of the 5th International Symposium on Memory Management (ISMM)*. 84–94.

[37] David Schwalb, Tim Berning, Martin Faust, Markus Dreseler, and Hasso Plattner. 2015. nvm malloc: Memory Allocation for NVRAM. *ADMS@ VLDB* 15 (2015), 61–72.

[38] Paul R Wilson, Mark S Johnstone, Michael Neely, and David Boles. 1995. Dynamic Storage Allocation: A Survey and Critical Review. In *Proceedings of the International Workshop on Memory Management (IWMM)*. Springer, 1–116.

[39] Kai Wu, Jie Ren, Ivy Peng, and Dong Li. 2021. ArchTM: Architecture-Aware, High Performance Transaction for Persistent Memory. In *Proceedings of the 19th USENIX Conference on File and Storage Technologies (FAST)*. 141–153.

[40] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steve Swanson. 2020. An Empirical Guide to the Behavior and Use of Scalable Persistent Memory. In *Proceedings of the 18th USENIX Conference on File and Storage Technologies (FAST)*. 169–182.