PMAlloc: A Holistic Approach to Improving Persistent Memory Allocation

ZHENG DANG and SHUIBING HE, Zhejiang University, Hangzhou, China
XUECHEN ZHANG, Washington State University Vancouver, Vancouver, USA
PEIYI HONG, ZHENXIN LI, XINYU CHEN, and HAOZHE SONG, Zhejiang University, Hangzhou, China
XIAN-HE SUN, Illinois Institute of Technology, Chicago, USA
GANG CHEN, Zhejiang University, Hangzhou, China

Persistent memory allocation is a fundamental building block for developing high-performance and in-memory applications. Existing persistent memory allocators suffer from many performance issues. First, they may introduce repeated cache line flushes and small random accesses in persistent memory for their poor heap metadata management. Second, they use static slab segregation resulting in a dramatic increase in memory consumption when allocation request size is changed. Third, they are not aware of NUMA effect, leading to remote persistent memory accesses in memory allocation and deallocation processes. In this article, we design a novel allocator, named PMAlloc, to solve the above issues simultaneously. (1) PMAlloc eliminates cache line reflushes by mapping contiguous data blocks in slabs to interleaved metadata entries stored in different cache lines. (2) It writes small metadata units to a persistent bookkeeping log in a sequential pattern to remove random heap metadata accesses in persistent memory. (3) Instead of using static slab segregation, it supports slab morphing, which allows slabs to be transformed between size classes to significantly improve slab usage. (4) It uses a local-first allocation policy to avoid allocating remote memory blocks. And it supports a two-phase deallocation mechanism including recording and synchronization to minimize the number of remote memory access in the deallocation. PMAlloc is complementary to the existing consistency models. Results on six benchmarks demonstrate that PMAlloc improves the performance of state-of-the-art persistent memory allocators by up to 6.4× and 57× for small and large allocations, respectively. PMAlloc with NUMA optimizations brings a 2.9× speedup in multi-socket evaluation and is up to 36× faster than other persistent memory allocators. Using PMAlloc reduces memory usage by up to 57.8%. Besides, we integrate PMAlloc in a persistent FPTree. Compared to the state-of-the-art allocators, PMAlloc improves the performance of this application by up to 3.1×.

This work was supported in part by the National Key Research and Development Program of China under Grant 2023YFB4502100, 2021ZD0110700, the National Science Foundation of China under Grant 62172361, the Major Projects of Zhejiang Province under Grant LD24F020012, the Program of Zhejiang Province Science and Technology under Grant 2022C01044, and the US National Science Foundation under CNS 1906541 and 2216108.

Authors' addresses: Z. Dang, S. He (Corresponding author), P. Hong, Z. Li, X. Chen, H. Song, and G. Chen, College of Computer Science and Technology, Zhejiang University, Hangzhou 310027, China; e-mails: dangzheng@zju.edu.cn, heshuibing@zju.edu.cn, hongpeiyi@zju.edu.cn, zhenxin@zju.edu.cn, xy.chen@zju.edu.cn, haozheshz@zju.edu.cn, cg@zju.edu.cn; X. Zhang, School of Engineering and Computer Science, Washington State University Vancouver, Vancouver, WA 98686; e-mail: xuechen.zhang@wsu.edu; X.-H. Sun, Department of Computer Science, Illinois Institute of Technology, Chicago, IL 60616; e-mail: sun@iit.edu.

ACM 0734-2071/2024/09-ART7
https://doi.org/10.1145/3643886

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

Additional Key Words and Phrases: Dynamic memory allocation, persistent memory, memory fragmentation, non-uniform memory access

ACM Reference Format:
Zheng Dang, Shuibing He, Xuechen Zhang, Peiyi Hong, Zhenxin Li, Xinyu Chen, Haozhe Song, Xian-He Sun, and Gang Chen. 2024. PMAlloc: A Holistic Approach to Improving Persistent Memory Allocation. *ACM Trans. Comput. Syst.* 42, 3–4, Article 7 (September 2024), 52 pages. https://doi.org/10.1145/3643886

1 INTRODUCTION

Dynamic allocation of persistent memory is heavily used for building high-performance applications from indexing structures [14, 50, 51, 56–59], transactional memory [19, 29, 31, 77], graph processing [37, 38, 75], to in-memory database systems [4, 16, 54, 63]. Memory allocators are usually well-tuned for volatile memory (e.g., DRAM) to achieve low latency, high scalability, and low fragmentation [6, 26, 67]. The adoption of persistent memory (e.g., Intel Optane DIMMs [17]) 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 with low overhead so they can safely recover allocated memory objects after failures.

Many allocators have been designed for persistent memory [8, 10, 23, 62, 63, 71]. To achieve efficient allocation and deallocation, they need to manage persistent heaps via various types of metadata (e.g., object bitmaps, slab structures, extent headers, and write-ahead logs). In this article, we name these auxiliary data structures *heap metadata*. For example, PMDK [17] and nvm_malloc [71] use bitmaps to mark objects that have been allocated. PAllocator [63] uses logs to ensure crash consistency between the user's application and its own operations. 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.

Our research shows that the existing persistent memory allocators only achieve suboptimal performance for the following four reasons: First, *their poor metadata management leads to cache line reflushes.* A typical size of CPU cache line is 64 B [63]. 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 7.5× higher than the latency of writes [13]. 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 3.2). Frequent cache line reflush operations cause the degraded performance of persistent memory allocators.

Second, *their poor metadata management leads to small random accesses in persistent memory.* Heap metadata of allocators tend to be randomly accessed in persistent memory. Many allocators (e.g., PMDK, PAllocator, and Makalu [8]) 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 [77, 79] 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.* 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 [76]. This segregation-induced fragmentation 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. This kind of fragmentation increases memory usage by up to 2.8× for workloads with changing allocation sizes and frequent "delete" operations (Section 3.4).

Fourth, *all of them ignore the* ***non-uniform memory access (NUMA)*** *effect.* Persistent memory is mainly deployed with multi-socket CPUs using the NUMA architecture at data centers. Prior works [46, 74, 80] reported that, compared to DRAM, the impact of NUMA is more pronounced for persistent memory. Allocators ignoring the NUMA effect may compromise application performance in both allocation and deallocation operations. Specifically, in the allocation operations, the memory blocks in the persistent heaps may be located in different NUMA nodes. The existing allocators may allocate the memory blocks belonging to remote NUMA nodes, resulting in a longer latency of the subsequent memory accesses. When releasing memory blocks, the deallocation operations running on one NUMA node may have to modify its metadata on another NUMA node. Our experimental results show that the persistent memory allocators ignoring the NUMA effect degrade the performance of applications by up to 4.6× (Section 3.5).

In this article, we introduce a NUMA-aware, high-performance, and fail-safe persistent memory allocator named PMAlloc. Our design objectives focus on the efficient elimination of cache line reflushes and the reduction of small random writes in heap metadata management. Additionally, we aim to mitigate slab-induced memory fragmentation and support NUMA-aware allocation and deallocation. To achieve these goals, PMAlloc supports the following optimizations:

First, PMAlloc 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. An interleaved appending method is also adopted in log-based structures (e.g., write-ahead logs) to eliminate reflushes for writing at log tails. Second, because in-place metadata updates cause random accesses in persistent memory with the limited write buffer size [79], 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, PMAlloc 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. Fourth, PMAlloc maintains a dedicated persistent memory heap for each processor core and adopts a local-first allocation policy to avoid allocating remote memory blocks. Furthermore, it supports a two-phase deallocation mechanism including *recording* and *synchronization* to minimize the number of remote memory accesses in the deallocation. Specifically, we call the node where a memory block is located its *owner node*. Each owner node uses both a regular bitmap and a shadow bitmap in DRAM for deallocation. In the recording phase, a remote deallocation request is only recorded in the shadow bitmap of its owner node of the requested memory block but not visible to users yet. When the regular bitmap does not have enough space to serve a request, synchronization will be triggered to apply the changes in the shadow bitmaps and make these blocks available.

PMAlloc currently supports both log-based and garbage-collection-based crash-consistency models.[1] Results on six benchmarks demonstrate that PMAlloc improves the performance of state-of-the-art persistent memory allocators by up to 6.4× for small allocations and 57× for large allocations. PMAlloc with NUMA optimizations brings a 2.9× speedup in multi-socket evaluation and is up to 36× faster than other persistent memory allocators. Using PMAlloc reduces memory usage by up to 57.8%. We also integrate PMAlloc in a persistent FPTree [64]. With PMAlloc, the performance of this application is improved by up to 3.1× compared with the state-of-the-art allocators. We further evaluate the performance of PMAlloc with six real-world applications and it outperforms other persistent memory allocators by up to 39×.

An earlier conference version of this article was presented in Reference [20]. Here, we extend the previous paper in several aspects. First, we highlight the critical need for a NUMA-aware persistent memory allocator and introduce NUMA-aware optimizations for both allocation and deallocation procedures. We provide exhaustive NUMA-related evaluations on PMAlloc, along with a performance comparison with existing allocators. Second, we enrich the discussion on programming semantics and safety, providing a detailed analysis of existing consistency models applicable to persistent memory allocators. We also discuss the applicability of our proposed techniques to future persistent memory products. Third, this article includes additional experiments, as well as more comprehensive numerical statistics and analysis.

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. They are typically configured to align with multiples of the page size, facilitating the rapid location of metadata for memory objects [39, 71].
- **Slabs** are pre-allocated extents in persistent memory and containers of fixed-size free blocks. The slab size is 64 KB in this article. 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) [63, 71]** 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).

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

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 [26] and nvm_malloc [71]). 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 PMAlloc. An *activated list* stores the VEHs of allocated extents. A *reclaimed list* stores the VEHs of freed extents with physical persistent memory 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, then its VEH will be moved to the activated list. If none is found, then a new VEH is created and added to the activated list. When an extent is freed, it is returned to the reclaimed list. PMAlloc uses a decay-based approach to manage free extents in the reclaimed list and retained list [26]. 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}, then 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, then 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.

Metadata association: Allocators require an indexing structure to associate metadata with the memory objects they manage. Typically, when users deallocate memory, they only provide the starting address of the memory object. The allocator then uses this index to retrieve the corresponding metadata, such as the vslab or the VEH. In this article, we adopt a page-based radix tree to serve this purpose, similar to jemalloc. Each 4 KB page managed by the allocator is represented by a 16-byte leaf in the radix tree, which results in a small overhead of 0.4% to the total heap size. The leaf node contains a size class field to identify whether the page is part of a slab or an extent, and a pointer pointing to either the vslab or the VEH. With the radix tree, the allocator can efficiently locate a memory object's metadata by aligning the object's starting address to the page size and then searching the tree. Note that when new memory regions are allocated by the operating system, their internal pages are initially registered in the radix tree before any allocations take place. This mechanism also enables the identification of illegal addresses in *free()* calls, as detailed in Section 4.4.

2.3 Consistency Models for Persistent Memory Allocators

In persistent memory allocators, system crashes can lead to a misalignment between the allocator's metadata and the actual memory blocks accessible to applications. Such inconsistencies arise when a memory block is allocated, but a system failure occurs before the user can store this block in a designated pointer. This phenomenon is referred to as "persistent memory leakage." Similar consistency issues may also arise during deallocation operations. Specifically, if the allocator

marks a block as released but the user has not nullified the associated pointer, then the allocator may re-allocate the same block, leading to unintended data corruption. Consequently, maintaining consistency of allocator metadata and user-accessible memory blocks is a primary concern for persistent memory allocators. Current designs of persistent memory allocators primarily adhere to three major consistency models:

Log-based model: In the log-based model [63, 71], users must supply the pointer for allocation or nullification through the allocator interface. This model employs WALs to record changes in both heap metadata and user pointers. In the event of a system crash, interrupted operations can be recovered to a consistent state by replaying the WALs. Within this model, allocators must use flush instructions to persist not only metadata changes but also the corresponding log entries in the WALs. Although this approach incurs a higher persisting overhead, it provides the highest level of consistency. PAllocator [63] and nvm_malloc [71] use this model.

GC-based model: The GC-based model [8, 10] utilizes a **garbage collection (GC)** mechanism to rebuild heap metadata after a system crash, thereby eliminating the need to persist metadata and WALs. This model relies on traversing the heap from pre-defined root pointers, enabling the GC process to identify memory blocks accessible to the user and reconstruct their metadata. Although this approach often yields superior allocation speeds, it presents several limitations. First, the GC-based model restricts certain capabilities of unmanaged-memory languages like C/C++, commonly used in the development of performance-sensitive systems [63]. Second, the garbage collection technique utilized in existing persistent allocators, specifically conservative garbage collection [8, 9], is error-prone and may not prevent memory leaks [10]. Makalu [8] and ralloc [10] apply this model.

Internal collection based (IC-based) model: The internal collection-based model [17] neither prevents memory leaks nor guarantees leak detection. Instead, it allows users to identify potentially leaked memory blocks by exposing the allocator's internal memory block collections through a dedicated interface. Users can then identify leaks by traversing this full set of blocks and comparing it against their own set of reachable memory blocks. This approach requires persisting allocator metadata. However, it eliminates the need for WALs, albeit at the cost of potential memory leaks and traversal. overhead. PMDK [17] adopts this model.

3 MOTIVATION

In this section, we begin by examining the potential overhead introduced by persistent memory allocators in real-world applications. We then experimentally investigate 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 Allocation Overhead in Persistent Memory Applications

The persistent memory allocators differ substantially from their volatile counterparts. Volatile memory allocators focus solely on providing fast, scalable memory allocation and deallocation. Differently, persistent memory allocators bear the added responsibility. They must ensure that memory operations are executed persistently and crash-consistently. This necessitates the inclusion of expensive flush and fence operations and additional data structures like WALs to maintain crash consistency, thereby making persistent memory allocators more time-consuming than volatile ones.

To prove that, we conducted experiments using a persistent B+-Tree, FPTree [64] and ran it with multiple existing persistent memory allocators (see Section 2.3). We collected the execution time of the benchmarks using the Linux *perf* tools. FPTree is executed with 40 threads. FPTree uses the same workload configuration as described in Section 8.5. Figure 1 shows the results. For IC- and

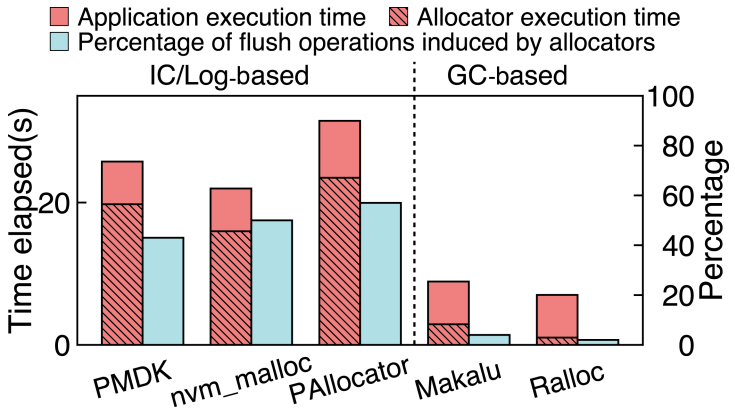


Fig. 1. Allocators' execution time and percentage of flush operations induced by allocators.

log-based allocators, the allocator's overhead can be exceedingly high, constituting up to 74.8% of the application's total execution time. This observation aligns with the previous research [46]. Furthermore, we found that flush operations initiated by the allocators can make up to 57% of the total flush operations. For GC-based allocators, Makalu and Ralloc account for 43.2% and 14.7% of the total execution time, respectively. They rarely use flush operations in the runtime, because they use a weak consistency model (discussed in Section 2.3).

The main reason behind this performance degradation is the consistency constraints imposed by persistent semantics. Allocators are compelled to immediately persist their metadata modifications using costly flush and fence operations. Worse, flushing small chunks of metadata can trigger cache line reflushes and result in random access to persistent memory. We will delve into these issues in Sections 3.2 and 3.3. In addition, if memory allocators are not designed for widely adopted NUMA architecture, then they can inadvertently cause remote memory accesses as discussed in Section 3.5. Given the considerable overhead associated with persisting allocator metadata, there is an urgent need to manage allocation metadata more efficiently.

3.2 Allocator-induced Cache Line Reflushes

With modern Intel processors, standard flush operations such as *clflushopt/clwb* are designed to return quickly once the flushed data arrives at the **write pending queues (WPQs)** in the processor's memory controller. Subsequent writes to persistent memory happen asynchronously, resulting in a low flush latency (approximately 100 ns). However, repeated flushes to the same CPU cache line induce latency. This is due to the requirement that data must first be read back into the cache for a new flush instruction to be executed. If the most recent data is still in transit within the WPQ, then the subsequent flush must wait until the preceding flush operation has been completed in the persistent memory and the data has been reloaded into the cache. This behavior unveils the inherent characteristics of persistent memory hardware, which has a high latency of reads and writes, consequently leading to a high latency of cache line reflushes.

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. This is because

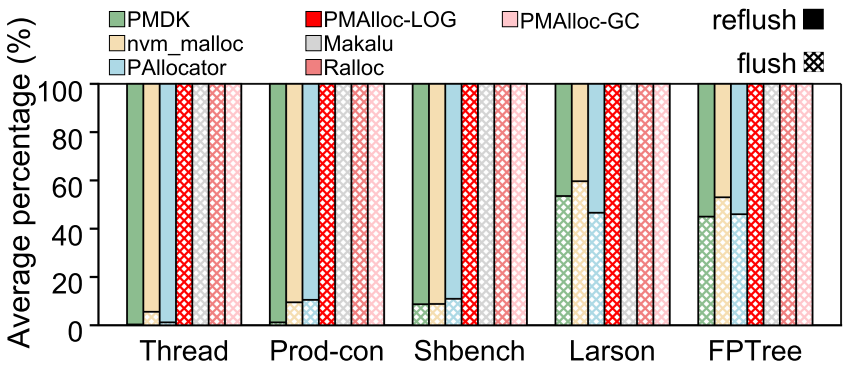


Fig. 2. Ratio of cache line reflush.

the preceding flush instruction has extra time to complete when the reflush distance is increased, thereby reducing the waiting time for any subsequent reflush instruction. In this article, 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 3× and 7× higher than random and sequential writes in persistent memory [13], 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 8. We trace flush operations in the benchmarks by substituting standard flush functions with custom C preprocessor macros. The percentage of both cache line reflushes and regular flushes are shown in Figure 2. 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. We also run FPTree with persistent memory allocators to show the impact of cache line reflush in persistent memory applications. The results show that the number of cache line reflushes accounts for 55%, 47%, and 54% for PMDK, nvm_malloc, and PAllocator, respectively. Makalu and Ralloc have zero reflushes, because they rarely using flush operations in the runtime. They ensure consistency through post-crash **garbage collection (GC)**, which leads to a longer recovery time and weaker consistency guarantee. Compared to these works, PMAlloc can eliminate cache line reflushes in both log-based (PMAlloc-LOG) and GC-based (PMAlloc-GC) consistency models.

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 1,000 flush operations of metadata when running the DBMStest benchmark [25] using four allocators, including nvm_malloc, PAllocator, PMDK, and Makalu. We exclude Ralloc because its open-source

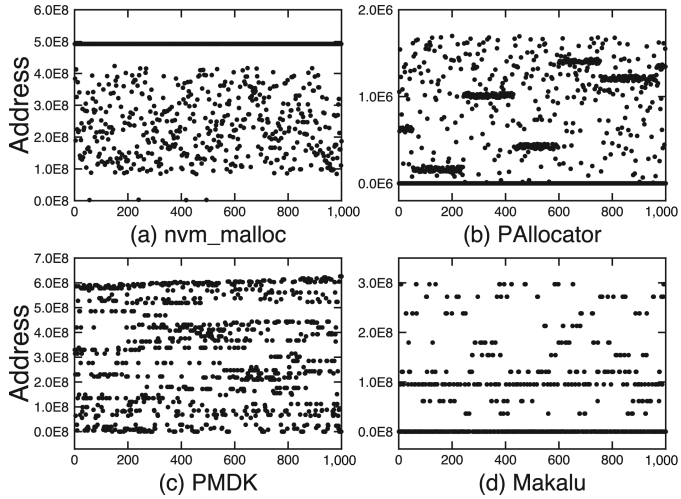


Fig. 3. Small random writes in large allocation. The X-axis denotes the number of flushes.

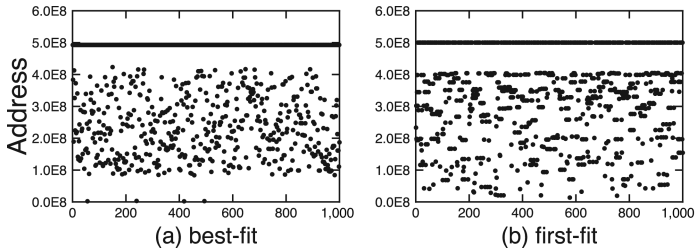


Fig. 4. Small random writes of nvm_malloc with different allocation algorithms.

implementation does not work correctly for large allocations. We show the results in Figure 3. 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 specialized allocation algorithms (i.e., best-fit, first-fit, or their variants). Their primary goal is to identify the most appropriate extents to minimize memory fragmentation. Because the optimal extent candidate can be located in any memory region within the heap space, random accesses are generated when updating its associated bookkeeping metadata. To further investigate the impact of different allocation algorithms, we modify the allocation strategy of `nvm_malloc` to implement both best-fit and first-fit algorithms. We execute the DBMStest benchmark using both implementations. As illustrated in Figure 4, the allocator's memory access pattern remains highly random. Regardless of the specific allocation algorithm, after a sequence of allocations and deallocations, the most favorable extents for serving neighboring requests can reside in disparate memory regions. This leads to small random accesses for updating bookkeeping metadata.

3.4 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 [26] and tcmalloc-2.9.1 [28]) and persistent memory allocators (e.g., Makalu [8], Ralloc [10], nvm_malloc [71], PMDK-1.11 [17], and PAllocator [63]). Slabs are

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 1,000–2,000 B

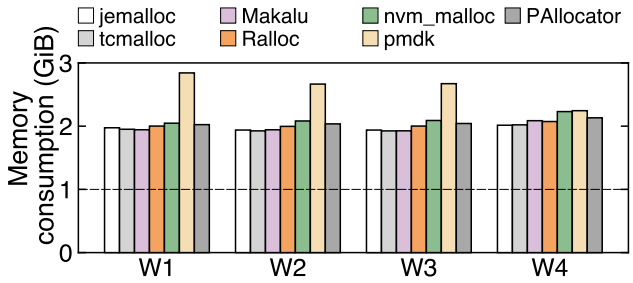


Fig. 5. Peak memory consumption of Fragbench.

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 could be changing over the execution lifespan of real-world server applications [40, 69, 76]. We run the fragmentation benchmark simulating the real-world behaviors of memcached storage systems in Meta [69] (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 [32, 40]. Similar workloads have been used in the prior research (i.e., RAMCloud [69], PAllocator [63], and log-structured NVMM [31]).

The peak memory consumption is presented in Figure 5. 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 [41]. 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 static slab segregation in persistent memory has a larger impact than in volatile memory, because the memory fragments cannot be eliminated by restarting.

3.5 Allocator-induced Slow NUMA Access

NUMA architecture has been widely used to increase the capacity and bandwidth of persistent memory. A NUMA system consists of multiple NUMA nodes, which further consist of CPU cores and DIMMs (DRAM/PM). The NUMA nodes are connected via inter-node links, e.g., Intel Ultra

[2]Although there are eight workloads in the original Fragbench, we only choose the four workloads, because other workloads show similar patterns.

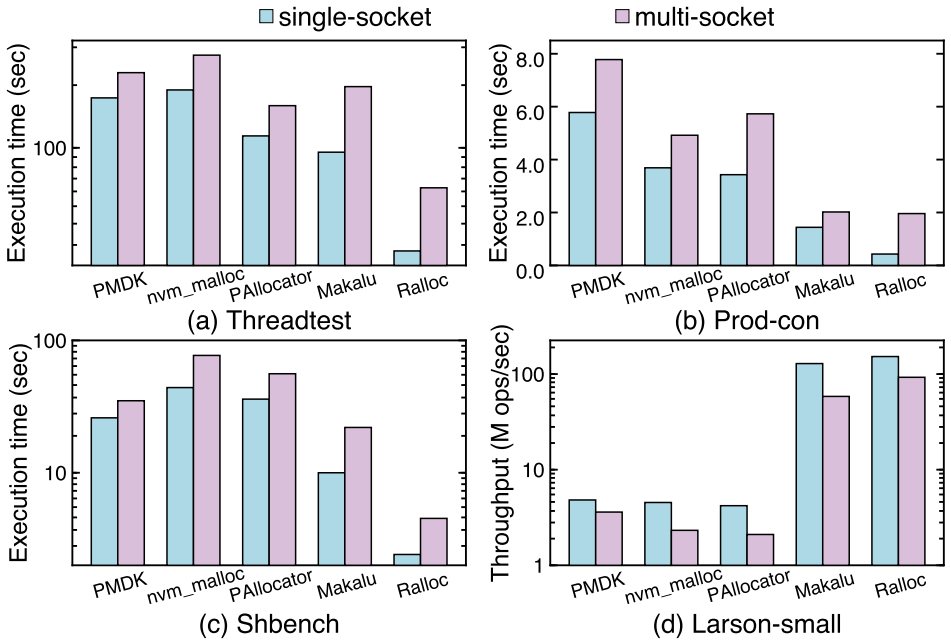


Fig. 6. NUMA effect with various persistent memory allocators.

Path Interconnect. Prior research [45, 74, 80] has established that accessing persistent memory located on remote NUMA nodes results in increased latency. This performance degradation is primarily due to the directory-based cache coherence protocol used in current Intel processor architectures [36] for NUMA domain management. In this protocol, the coherence state of each cache line is embedded within the line itself in the persistent memory. Consequently, remote memory accesses may necessitate an additional directory "write" operation to update the coherence state, for instance, transitioning from an "Exclusive" to a "Shared" state. The directory "write" operation can be triggered by either remote read or write access. Prior work [46] has demonstrated that this mechanism can substantially reduce remote read bandwidth, as it involves both read and write operations during read accesses.

We quantitatively study the NUMA impact using four established benchmarks and multiple persistent memory allocators. These benchmarks are executed with 20 threads, either confined to a single NUMA node (single-socket configuration) or distributed across both nodes (multi-socket configuration). In the multi-socket setup, we use *pthread_setaffinity_np()* to pin half of the threads on the first node and the other half on the second node. Figure 6 presents the results. Across all multi-socket experiments, we observe that existing allocators experience performance degradation, varying between 1.29× to 4.6×. This degradation arises due to the intricacies of NUMA architecture influencing both memory allocation and deallocation operations. During allocation, threads may serve memory blocks from a remote NUMA node, introducing added latency for subsequent memory operations. During deallocation, releasing a memory block on remote NUMA nodes leads to write amplification for updating metadata (e.g., bitmaps) on the remote NUMA nodes.

In summary, both our study and prior work [46] show that NUMA-aware persistent memory allocators are highly needed for developing large-scale high-performance applications.

4 PMALLOC

In this section, we present the programming model of PMAlloc and describe the design of its major components: small allocator and large allocator. The PMAlloc software is developed with four optimizations including (1) interleaved mapping, which reduces cache line reflushes, (2) slab morphing, which alleviates segregation-induced fragmentation, (3) log-structured bookkeeping, which improves the write locality, and (4) NUMA-aware allocations and deallocations for the persistent memory system with multiple NUMA nodes. We illustrate all the components of PMAlloc and where each optimization is applied in Figure 7.

4.1 Programming Model

We use `pmalloc_init()` to create a new PMAlloc instance and `pmalloc_exit()` to safely exit. To avoid the memory leak, we adopt the `pmalloc_malloc_to()` and `pmalloc_free_from()` API used in other allocators [17, 63, 71] to atomically allocate and free objects on persistent memory, respectively. Function `pmalloc_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 [10, 12, 15]. The `pmalloc_free_from()` returns a block or an extent specified by *address* to the persistent memory heap and nullifies the user-provided persistent pointer.

Currently, we implement two variants of PMAlloc including PMAlloc-LOG supporting log-based transactional model and PMAlloc-GC supporting GC-based model (see Section 10). PMAlloc-LOG uses **write-ahead logs (WALs)** to maintain crash consistency. When an allocation or deallocation action is initiated, all associated metadata updates are recorded in the WALs. These logs also contain the corresponding user-provided pointers passed via the allocation or deallocation interfaces. Before updating the actual data structures in persistent memory, PMAlloc-LOG ensures that the WALs are written persistently. During failure recovery, PMAlloc-LOG utilizes the WALs to identify any incomplete allocation or deallocation operations, whose metadata have been updated but user-provided pointers have not. For such partially completed operations, PMAlloc-LOG reverses the metadata changes to prevent any leakage in the persistent memory. In PMAlloc-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 based on user-defined root pointers. The GC blocks the normal execution of applications [8]. PMAlloc provides a `pmalloc_set_root()` interface for users to specify a persistent pointer as the root pointer. The root pointers are stored in a specific persistent space, with up to 512 top-level roots per arena. For large allocations, PMAlloc-GC has the same code path as PMAlloc-LOG.

4.2 Small Allocator

For small allocation (<16 KB), PMAlloc 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 that has the least number of assigned threads. An arena 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, then the working thread will fill it until full using slabs from their corresponding $freelist_{slab}$ in the

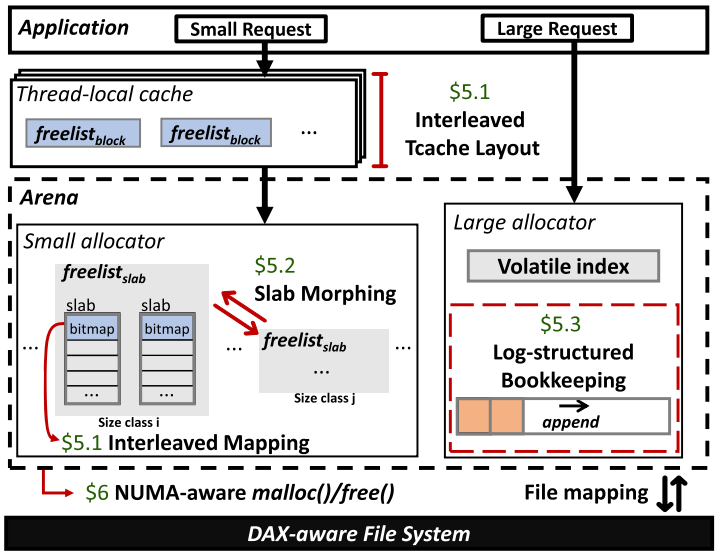


Fig. 7. Overview of PMAlloc.

arena. Thread synchronization is required here, because multiple threads may be attached to the same arena. If there is no slab in $freelist_{slab}$, then 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 a radix-tree (as detailed in Section 2.2) to find its size class and *vslab*. The *vslab* contains a pointer to the corresponding slab header in persistent memory. The persistent slab header will be updated to record the release. Then, the working thread will try to return the block to its corresponding tcache for serving future allocation requests. If the $freelist_{block}$ of the tcache is full, then the working thread will return the small block to its *vslab* directly, bypassing its tcache.

PMAlloc 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 PMAlloc is responsible for allocating slabs and extents that are ranging from 16 KB to 2 MB. For objects larger than 2 MB, PMAlloc calls `mmap()` to allocate a given size extent. The architecture of the large allocator is shown in Figure 11 in Section 5.3. When `pmalloc_malloc_to()` is called, it first searches the reclaimed list using the best-fit algorithm. If no extent is found, then the search is repeated using the retained list. If an extent is found, then 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. The suitable part after splitting is returned to the user, while the remaining part is reinserted into the source list. If no extents are available in either the reclaimed list or the retained list, then PMAlloc calls `mmap()` to allocate a new extent of 4 MB, which is split into two parts. PMAlloc 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, PMAlloc adds an item to the radix-tree pointing to the VEH for the part.

When `pmalloc_free_from()` is called to free a large memory object, PMAlloc searches for its VEH in the radix-tree using its memory address. The VEH is moved from the activated list to the reclaimed list. If the extents adjacent to the currently freeing extent are also free, then they will be coalesced into a new, larger extent before inserting to the reclaimed list. PMAlloc uses a decay-based approach to manage VEHs in the 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 Sanity Check

Unlike volatile main memory, simple programming bugs can cause permanent corruption to persistent memory heap [23]. Therefore, it is imperative for a robust persistent memory allocator to proactively identify and mitigate such adverse behaviors. In PMAlloc, we have added layers of sanity checks to safeguard against incorrect or malicious use of memory allocation APIs, specifically targeting issues such as double-free or invalid-free. We describe three sanity-check mechanisms.

Mapped address verification: Initially, all users' address spaces that are mapped to the persistent memory are registered in a radix tree maintained by PMAlloc. Upon a deallocation request, we check whether the address falls within the pre-registered address range in the radix tree. If it does not, then the radix tree generates an error for the unfound address, marking the deallocation request as illegal and promptly aborting it.

Internal range validation: If an address does fall within the registered address range, then PMAlloc subsequently verifies its validity against the address range of its corresponding *vslab* or VEH. For small blocks, the address must reside within the data space managed by the *vslab* and align precisely with the boundaries of memory objects within the slab. Conversely, for large extents, the address must correspond exactly to the extent start address recorded in the VEH.

Allocation status check: Even if an address passes the above sanity checks, one final check is performed to confirm whether the object at that memory location is currently allocated. The allocation status is verified using *vslab* for small memory objects and using VEH for large ones. This prevents the issue of attempting to deallocate an object that has already been freed, commonly known as a double-free error.

These rigorous checks form an integral part of PMAlloc's design, serving as preventive measures to guard against both unintended and malicious misuse of the memory allocator APIs.

5 OPTIMIZATION OF METADATA MANAGEMENT

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

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 [8, 10] 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 [23]. 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

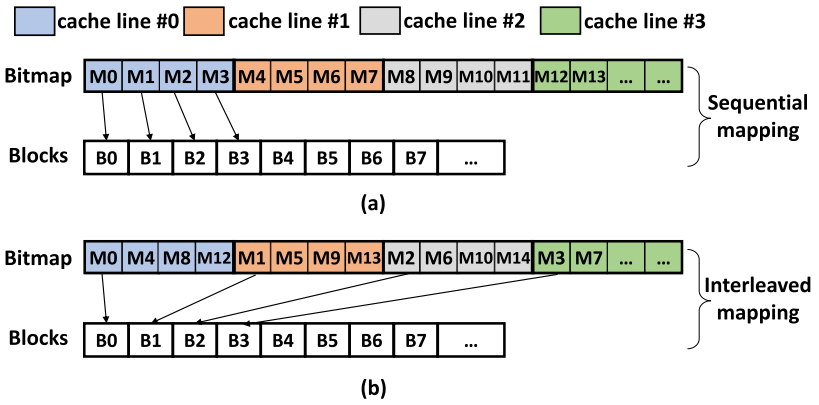


Fig. 8. Interleaved bitmap mapping.

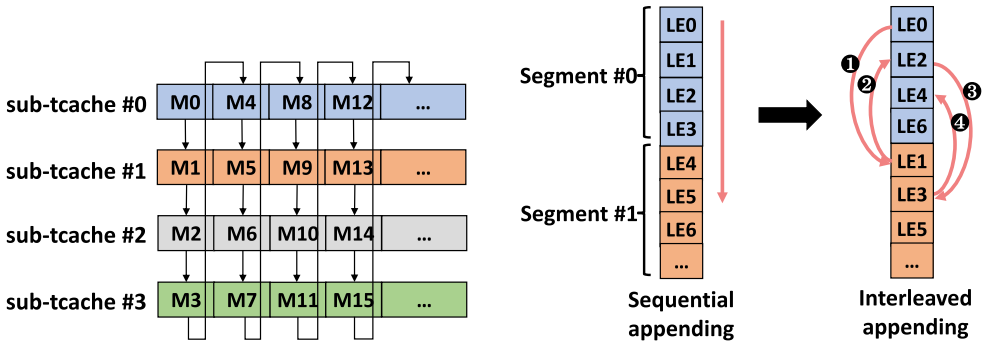
possible. Third, blocks in tcache may still be mapped to the same cache line. Therefore, none of the existing work completely solves the problem.

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 8 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 lines #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 9(a)). 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 9(a). 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.

Interleaved appending of logs. Existing approaches sequentially append new log entries to the tail of WALs or other log-based structures as shown in Figure 9(b). They may cause cache line reflushes if the size of log entries is smaller than the cache line size. To eliminate this kind of cache line reflushes, we design an interleaved log-appending approach. Specifically, PMAlloc partitions



(a) Interleaved tcache layout. (b) Interleaved log appending.

Fig. 9. Interleaved mapping used in tcache and log structures.

the tail part of a log into multiple log segments (e.g., Segment #0,#1 in the figure). The size of each segment is 64 B. The segments are aligned to the cache line boundary. The log entries are appended to different segments alternatively. We use an example to further illustrate it as shown in Figure 9(b). The working thread appends five log entries denoted as *LE*0 to *LE*4. We assume their size is 8 B. When the sequential appending is used, *LE*1 to *LE*4 are appended to the log sequentially, causing cache line reflushes. In contrast, when the interleaved appending is used, *LE*1 and *LE*3 are appended to Segment 1. *LE*2 and *LE*4 are appended to Segment 0. Because these consecutive log entries (e.g., *LE*0 and *LE*1) are stored in two segments in different cache lines, PMAlloc can effectively eliminate repeated cache line reflushes.

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, PMAlloc 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. PMAlloc will choose a slab for morphing and transforming its metadata.

Selecting a slab candidate for morphing. PMAlloc 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 8.7). 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, PMAlloc 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

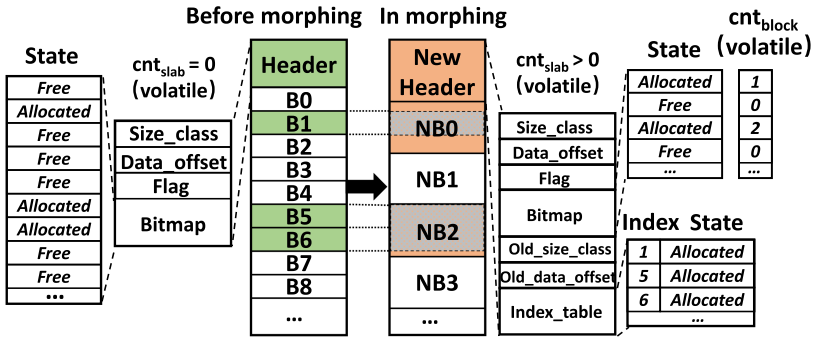


Fig. 10. Illustration of slab morphing.

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* that comprises entries for each $block_{before}$. Each entry within the *index_table* stores the block index in $slab_{before}$ and the current allocation state of the block. All allocation states are initially set to "Allocated," and they are subsequently updated to "Free" once $block_{before}$ is deallocated. The presence of the *index_table* is crucial for maintaining the recoverability of $block_{before}$, since direct access to the bitmap of $slab_{before}$ is no longer possible. 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 $cnt_{slab} > 0$, then 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 corresponding bitmap bit for a block will remain set if its cnt_{block} is not equal to zero. This non-zero value indicates that the memory block is still occupied by one or more $block_{before}$.

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 10 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, PMAlloc 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 procedure introduces a small overhead, because it only involves metadata modification. This overhead could be further reduced by employing a background thread to identify and morph slabs with low memory utilization. However, for implementation simplicity, this optimization is not incorporated in the current version. 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 $blocks_{before}$, PMAlloc needs to modify its state in the index_table and flush it. 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 8.3.

5.3 Log-structured Bookkeeping

For large allocation and release, PMAlloc 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 11. 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 [63], and (2) our address is 4 KB-aligned, thus the lower 12 bits are not needed in the log entry. For processors supporting the 5-level page table processor feature [33], they allow the operating system to extend the size of virtual addresses from 48 bits to 57 bits. In such systems, we utilize a log entry of 16 B. The first 8 B stores "addr," and the second 8 B stores "log type" and "size." Despite this increment, the log entry size remains relatively insignificant compared to the requested allocation size. 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 uniform 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 12. At the time of initialization, PMAlloc creates a file of 100 MB in persistent memory to store log entries. A log file is divided into chunks of 1 KB, each 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.

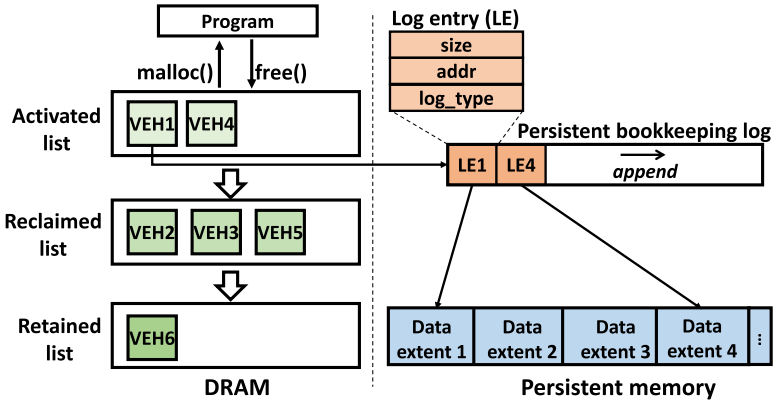


Fig. 11. Illustration of log-structured bookkeeping.

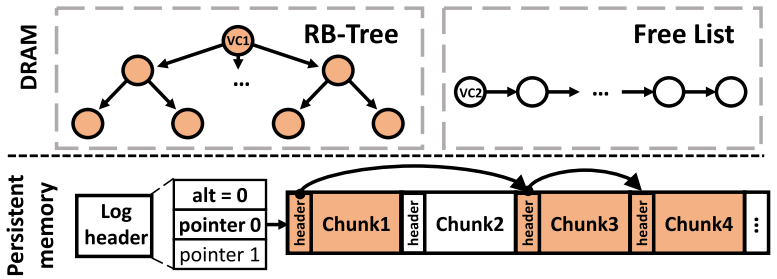


Fig. 12. The memory layout of the persistent bookkeeping log. VC denotes vchunk. Chunks in orange and white colors denote active and free chunks, respectively.

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, PMAlloc uses a red-black tree to manage the vchunks of the allocated chunks and a free chunk to manage free 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, then a new chunk is created and appended to the tail of the log file in persistent memory.

Basic log operation. In PMAlloc, 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 GC to drop the log entries that are marked as deleted by the tombstones. PMAlloc supports two GC algorithms, including *fast GC* and *slow GC* [31], which are designed to make different tradeoffs between the GC overhead and memory efficiency. Specifically, if the size of log files is larger than a certain memory usage threshold $Usage_{pmem}$ (default value of 0.2%), then PMAlloc first executes a fast GC. After that, if the size of log files still exceeds the threshold, then it applies a slow GC.

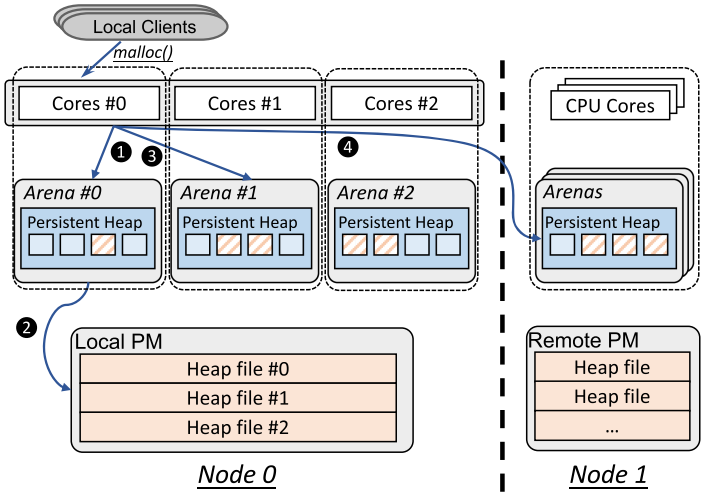


Fig. 13. NUMA-aware allocation process. Code path 1: allocation using the local arena. Code path 2: expand the local arena via OS. Code path 3: allocation using the arena on the same NUMA node. Code path 4: allocation using remote arenas on different NUMA nodes.

The fast GC algorithm scans the bitmap of each vchunk in the red-black tree. If the bitmap of a vchunk is empty, then it will be moved to the free list. Because the fast GC algorithm does not need to access persistent memory, its overhead is trivial.

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, PMAlloc marks $list_{new}$ as the current active chunk list by flipping the *alt* bit. Then it recycles all the chunks in $list_{old}$.

6 NUMA-AWARE MEMORY ALLOCATION AND DEALLOCATION

Contemporary computer systems usually use the popular NUMA architecture to improve their performance. To accommodate the NUMA architecture, PMAlloc should be carefully designed considering the longer latency of memory accesses across NUMA nodes. We need to enforce failure consistency while minimizing the number of remote memory accesses in the allocation and deallocation of persistent memory. PMAlloc uses the slab-based and extent-based (de)allocation for small and large blocks, respectively. In the following subsections, we first describe how we optimize the allocation and deallocation processes of small memory blocks in PMAlloc. Then, we describe how we adapt these optimizations to large (de)allocations.

6.1 NUMA-aware Allocation of Small Blocks

To reduce remote memory accesses across NUMA domains, PMAlloc needs to allocate local memory blocks first before allocating remote memory blocks in other NUMA nodes. For this purpose, PMAlloc creates a dedicated persistent memory heap for every *arena* corresponding to each CPU core as shown in Figure 13. Specifically, we first create heap files in DAX file systems mounted to different NUMA nodes. Then, when an arena is created, its heap space is mapped to the heap files based on the ID of the NUMA node that the arena is affiliated with. For example, as Figure 13

ALGORITHM 1: NUMA-aware allocation of small blocks

```
   Input: memory block size size_req
   Output: memory address addr_ret
1  SZ ← Get_Sizeclass_with_Size(size_req);
2  if not Try_to_alloc_with_tcache(tcache.freelist_block[SZ],addr_ret) then
       /* Try to fill tcache with the arena it is affiliated to.                    */
3      success_flag ← Try_Fill_Tcache(Get_Arena(tcache));
       /* If fails, try with arenas in the same node.                               */
4      if not success_flag then
5          foreach arena_local in the local node do
6              success_flag ← Try_Fill_Tcache(arena_local);
7          end
8      end
       /* If fails, try with arenas in remote nodes.                                */
9      if not success_flag then
10         foreach arena_remote in the remote node do
11             success_flag ← Try_Allocate_without_Tcache(arena_remote, addr_ret);
12             if success_flag then return addr_ret;
13         end
14     end
15     if not success_flag then
16         ERROR("Memory exhausted.");
17     end
       /* Now, we can use tcache to allocate the block.                             */
18     addr_ret ← Allocate_with_tcache (tcache.freelist_block[SZ]);
19 end
   /* Persist the allocation.                                                       */
20 Set_Bitmap(pslab.bitmap, addr_ret);
21 Flush(pslab.bitmap);
22 return addr_ret;
```

shows, Arena #0 is affiliated with Core #0. Therefore, the heap space of Arena #0 is mapped to the heap file #0 on NUMA node 0.

We design a NUMA-aware allocation policy to minimize the allocation from remote NUMA nodes. Its algorithm is shown in Algorithm 1. Its input is request size $size_{req}$, and its output is the allocated memory address $addr_{ret}$. Specifically, the working thread will allocate memory blocks from its tcache (Line #2). If the tcache is empty, then PMAlloc needs to fill the tcache with the arena that it is affiliated to (Line #3). PMAlloc will search the slabs of the given size class to find free memory blocks and mark them as allocated in the bitmap of *vslab* (i.e., the volatile header of a slab). If the given arena does not have enough space to fulfill the allocation request, then the working thread will first try to extend the heap space of the arena by creating and mapping new heap files through OS. If the OS request fails, then the working thread will turn to other *arena*s to search for suitable memory blocks. In this case, it will first check the *arena*s on the same node (Lines #4–8). If all of the previous functions fail, then the working thread eventually has to check *arena*s on other NUMA nodes to serve the request (Lines #9–14). The *Try_Allocate_Without_Tcache()* function will persistently allocate the memory block in the remote arena. Finally, $addr_{ret}$ is assigned with the address of the memory block to be allocated (Line #18) and PMAlloc will persist the allocation in the bitmap of *pslab* (i.e., the persistent header of a slab) in persistent memory (Lines #20 and 21).

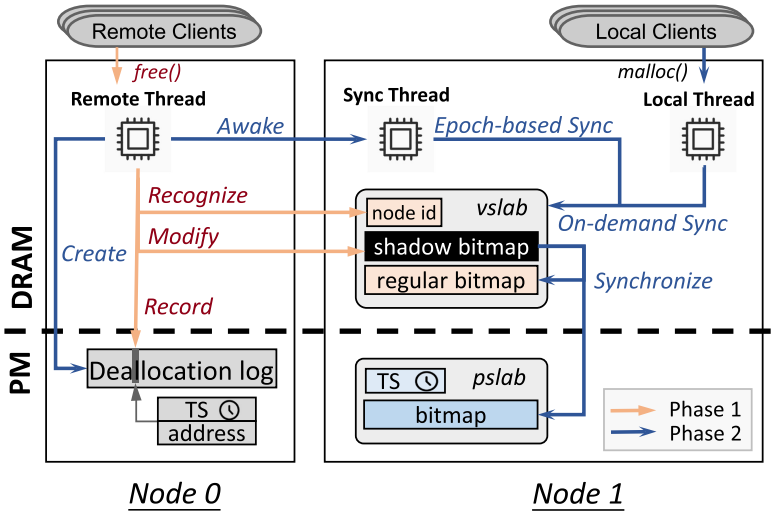


Fig. 14. Major data structures and operations of the two-phase deallocation mechanism in PMAlloc. *TS* is short for timestamp.

6.2 NUMA-aware Deallocation of Small Blocks

There are many challenges in the design of NUMA-aware deallocation. (1) Allocators have no knowledge of the whereabouts of incoming deallocation requests. Therefore, they have to serve both remote and local deallocation requests efficiently. (2) Allocators need to make sure the blocks released by remote threads can be reused for serving requests from local threads. Otherwise, *blowup* fragmentation [6] (i.e., any memory freed through remote deallocations cannot be reused again) may be induced. (3) The deallocation process used in the existing persistent memory allocators need many remote reads and writes for searching and updating the metadata of memory blocks in the arena, resulting in performance degradation of the allocators.

An intuitive approach to address these issues is creating a dedicated background thread for each NUMA node to handle deallocation requests from both local and remote threads. The background threads can complete the deallocation process without remote memory access. However, this approach does not work for persistent memory allocators, because it may induce an unacceptable synchronization overhead. This is because the calling threads cannot return immediately after handover to the background threads. They must wait until the metadata of the released memory blocks is persisted in persistent memory to avoid consistency problems.

In PMAlloc, we design a two-phase deallocation mechanism to minimize the number of memory accesses across NUMA nodes and the synchronization overhead. Its deallocation includes a *remote recording* phase and a *local synchronization* phase. Figure 14 illustrates the major data structures and steps of the deallocation process.

Data structures: *Vslabs* maintain NUMA node ID of a memory block, a *regular bitmap* whose bits denote the state of memory blocks between synchronizations, and a *shadow bitmap* whose bits denote the state of memory blocks that have been freed by remote threads but have not been synchronized with the regular bitmap. Each tcache also has a node ID, which is assigned by the thread that it belongs to. In persistent memory, PMAlloc has *pslabs* corresponding to vslabs and manages a *deallocation log* on each NUMA node. The pslab contains a *persistent bitmap* to record the state of memory blocks persistently and a **timestamp (TS)** to record when the persistent bitmap was modified. We utilize *_rdtsc()* instruction to obtain the hardware clock and generate TS,

ALGORITHM 2: NUMA-aware deallocation of small blocks

```
   Input: memory address addr_input to be released
1  vslab ← Search_Rtree_for_vslab(addr_input);
2  arena ← getArena(vslab);
   /* Step 1, identify remote deallocation.                              */
3  if vslab.nodeid ≠ tcache.nodeid then
       /* Step 2, record the release operation with deallocation log.    */
4      if deallocation_log.logsize == MAX_LOG_SIZE then
5          foreach arena_remote on remote nodes do
6              Create_or_Awake_thread(arena_remote, Epoch_based_Sync);
7          end
8          deallocation_log = Create_New_Log();
9      end
10     Append_Log_Entry_with_TS(deallocation_log);
11     Flush(deallocation_log);
       /* Step 3, register the block in shadow bitmap.                   */
12     vslab.has_shadow_bitmap ← TRUE;
13     Set_Bitmap (vslab,addr_input);
14 else
       /* Otherwise, perform local deallocation.                         */
15     SZ ← Get_Sizeclass(vslab);
16     if not Try_to_release_with_tcache(tcache.freelist_block[SZ],addr_input) then
17         Unset_Bitmap(vslab.regular_bitmap, addr_input);
18     end
       /* Persist the deallocation.                                      */
19     Unset_Bitmap(pslab.bitmap, addr_input);
20     Flush(Pslab.bitmap);
21 end
22 return;
```

and we use the ORDO primitive [44] to ensure correct ordering of timestamps across NUMA nodes. The deallocation log records remote deallocation operations locally. Its log entry consists of a TS to record the time of the logged operation and the address of the memory block to be deallocated. TS is used to ensure crash consistency when serving concurrent remote deallocations.

The two-phase remote deallocation procedure comprises the remote recording phase and the local synchronization phase. The remote recording phase (*Phase 1* in Figure 14) is executed by a deallocation thread. The thread first determines if the memory block being released is from a remote NUMA node (*Recognize* step). If so, then the thread first stashes this deallocation in the associated shadow bitmap of the block of the remote node (*Modify* step). Then, it records essential data for failure recovery in the local deallocation log (*Record* step). Finally, the local synchronization phase (*Phase 2*) is conducted either by allocation threads through an *On-demand Synchronize* mechanism or by dedicated synchronization threads through an *Epoch-based Synchronize* mechanism on remote nodes. A detailed description of these two deallocation phases is provided below.

Remote recording phase: Algorithm 2 shows the procedure of the NUMA-aware deallocation and its remote recording phase in PMAlloc. The algorithm takes the memory address $addr_{input}$ to be released as its input. The algorithm needs to check whether a request requires remote

deallocation (Line #3). It compares the NUMA node ID of the deallocation thread and the memory block to be deallocated to determine that. For the non-remote deallocation, it returns the memory block to tcache and updates its metadata in vslab and pslab (Lines #15–20). For the remote deallocation, PMAlloc only records the deallocation request in the deallocation log (Lines #4–11). Then, it sets the bits corresponding to blocks being deallocated in the shadow bitmap of its corresponding vslab in the remote NUMA node (Lines #12 and 13). PMAlloc does not update the regular bitmap and the persistent bitmap until the local synchronization phase. The pseudo-code of local synchronization is not displayed in Algorithm 2, because it is executed asynchronously and not in the critical path of deallocation. The whole remote recording phase is completed by deallocation threads without memory access to persistent memory on different NUMA nodes.

Local synchronization phase. In this phase, threads will synchronize shadow bitmaps and regular bitmaps and free the space in the deallocation log. PMAlloc supports two kinds of local synchronization: on-demand synchronization and epoch-based synchronization. Both synchronization processes are protected under a per-slab *mutex* to avoid conflict with other threads.

On-demand synchronization: it happens in the allocation process. When the allocation thread needs memory blocks to fill its tcache, it will search vslabs one-by-one for available blocks. Within each vslab, it will search the regular bitmap first. The block status in the shadow bitmap is invisible to the working threads if the number of available blocks in the regular bitmap is enough to fill the tcache. If they are not enough, then the on-demand synchronization will be triggered to synchronize the shadow bitmap of the current vslab. In the procedure of on-demand synchronization, the working thread (1) applies the changes recorded in the shadow bitmap to the regular bitmap, (2) applies the changes to the persistent bitmap, and (3) updates the TS in the pslab using the RDTSC instruction atomically. After the synchronization, the working thread can fill its tcache using the newly freed blocks through the regular bitmap. The corresponding log entries in the deallocation log will be freed and recycled in the epoch-based synchronization.

Epoch-based synchronization: it is triggered when the size of any deallocation log exceeds a threshold (e.g., 4 MB in our design) in the remote recording phase. The trigger thread will create a new deallocation log to handle future remote deallocation requests and awaken the background synchronization thread for each arena in the remote NUMA nodes. Each arena is assigned a dedicated synchronization thread that awaits the epoch-based synchronization signal. The synchronization thread will find all vslabs that have not been synchronized by on-demand synchronization in the arena. For each vslab, it synchronizes the shadow bitmap to the regular bitmap and persistent bitmap and then updates the TS in the pslab. When all synchronization threads complete their work, the trigger thread will be informed to recycle the previously used *deallocation log*. The garbage collection procedure is simply removing the previously used deallocation log entirely, because all the remote release operations recorded in the log are completed by synchronization threads.

Semantic and consistency guarantee. PMAlloc assumes the same programming semantics for deallocating remote memory blocks as it does for local ones. Our two-phase deallocation process is designed to be transparent from the user's perspective. In this design, the deallocation function call returns as soon as the remote recording phase is complete, while the synchronization phase is carried out asynchronously by background or allocating threads at a later time. Despite this separation in time and operation, PMAlloc guarantees the consistency of remote deallocations. We address this in two aspects:

First, PMAlloc strictly avoids allocating any memory block that has not been fully and persistently deallocated. This ensures that no memory blocks will be erroneously marked as released following a failure and subsequent recovery. With our two-phase deallocation mechanism, blocks that are in the process of being deallocated are stashed in shadow bitmaps. These blocks are

invisible to allocation threads until they have been successfully synchronized to persistent memory. Once this synchronization is complete, the blocks are entirely deallocated. Just as using the standard deallocation procedures, they are released in both the DRAM and persistent memory by changing the metadata on their resident NUMA node.

Second, PMAlloc guarantees that all delayed deallocations will be completely finalized once the deallocation function returns, even if a system failure occurs. This is achieved by using local deallocation logs. Although the metadata for deallocated blocks is not immediately updated when the deallocation function is called, the deallocation is logged by the deallocation thread. This log is later synchronized to the block's metadata during a recovery phase.

By adhering to these guidelines, PMAlloc ensures both semantic transparency and consistency, thereby offering a reliable solution for remote memory block deallocation in persistent memory systems.

6.3 NUMA-aware Optimizations for Large (De)Allocations

PMAlloc adapts the similar approaches to small (de)allocations for large (de)allocations. (1) Similar to small allocations, PMAlloc allocates local memory extents first before allocating remote memory extents in other NUMA nodes in the implementation of large allocation. (2) Similar to small deallocations, PMAlloc uses a two-phase extent deallocation mechanism to minimize the number of memory access across NUMA nodes. (3) PMAlloc uses deallocation logs to enforce crash consistency of large allocators.

NUMA-aware large allocations. When an allocation thread needs one large extent to serve user requests, it traverses the existing arenas to find one having enough space for the extent. The traversing order of arenas is similar to small allocations: the arena it is affiliated to, arenas on the local node, and then the arenas on the remote nodes. In this way, PMAlloc ensures that local memory extents are always allocated in preference to remote ones. When a candidate extent is chosen, the allocation thread will move its VEH to the activated list and append a log entry to the bookkeeping log to persistently record the allocation.

NUMA-aware large deallocations. PMAlloc also adopts a two-phase NUMA-aware deallocation approach, which also includes the remote recording phase and local synchronization phase, for serving remote release of large memory extents. Figure 15 shows the major data structures and operations in it. Similar to small deallocations, we add a *deallocation log* in each NUMA node to ensure crash consistency and a *shadow log* in each arena to temporarily store the tombstone log entries of remote released extents. We also add a *remote-released list* for each arena to record which VEHs point to the extent being remotely released.

— *Remote-recording phase.* The release thread records the deallocation operation in the deallocation log and appends a tombstone log entry for the released extent to the shadow log. Then it moves the corresponding VEH from the activated list to the reclaimed list and remote-released list.

— *Local synchronization phase.* PMAlloc also supports two kinds of synchronization approaches, on-demand synchronization and epoch-based synchronization. For on-demand synchronization, when an allocation thread finds an extent meeting its size requirement, it will check whether the VEH of this extent has been added to the remote-released list. If so, then it will migrate all the tombstones stored in the shadow log to the bookkeeping log and then remove all the VEHs from the remote-released list to ensure all the remote releases that happened before in the arena are synchronized. For epoch-based synchronization, it happens when any deallocation log for large deallocations exceeds the capacity threshold (4 MB). The synchronization threads will be awoken in each arena of remote nodes to

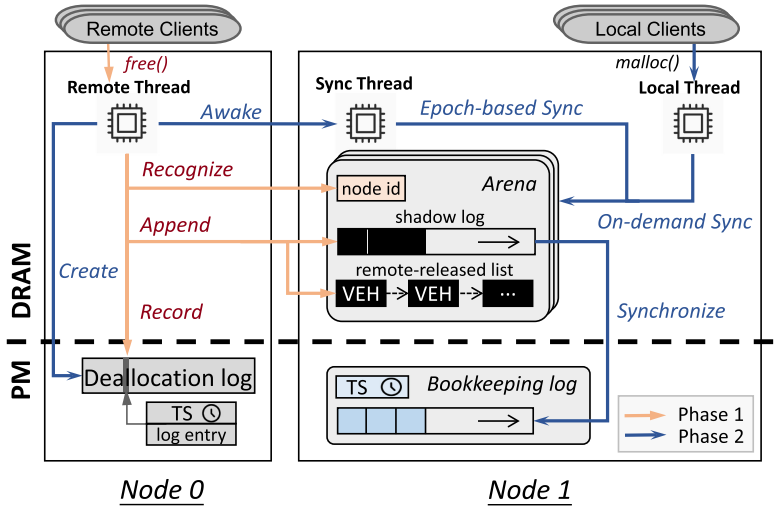


Fig. 15. Illustration of NUMA-aware large deallocation design. *TS* is short for timestamp.

synchronize the shadow log and empty the half-released list. After all the synchronization threads complete their work, the previously used deallocation log will be removed.

To ensure crash-consistency, PMAlloc also adds TS to deallocation log entries and persistent bookkeeping logs. The TS in a bookkeeping log will be updated when tombstone log entries are appended to it in the synchronization phase. Similarly to small deallocations, if a power failure happens before normal exits, then the recovery thread will traverse all the deallocation logs and replay the log entries whose TS is newer than its corresponding bookkeeping log. For these log entries, a tombstone of the remotely released extent will be appended to the bookkeeping log.

6.4 NUMA-aware Tcache Optimization

Utilizing tcaches can accelerate memory allocations by storing blocks being recently freed in the local thread. This approach has been adopted by most of the existing allocators [8, 10, 26, 28]. This section details how PMAlloc ensures NUMA-locality while using the tcache technique, even when faced with scenarios like thread migration.

Guaranteeing local memory items in tcache. Except in scenarios involving thread migration, PMAlloc guarantees that tcache only contains local memory items. First, for allocation operations, memory blocks are primarily sourced from local arenas. The only exception arises when local heaps are exhausted, an occurrence that is exceedingly rare. In such instances, the memory blocks allocated from remote heaps will bypass the tcache and are directly provided to the requesting thread. This ensures that the tcache remains populated solely with local memory items. Second, for deallocation operations, any remote deallocations are immediately identified and executed bypassing the tcache, ensuring tcache remains unaffected by remote deallocations.

Addressing OS-level thread migration. The operating system may silently migrate threads across NUMA nodes for efficiently utilizing hardware resources. However, existing persistent memory allocators do not adapt their allocation strategies to the OS-level thread migration. As a result, their tcache may continue to hold and allocate items from the original NUMA node, leading to remote object allocations. To address this, PMAlloc introduces an optional periodic thread migration monitoring mechanism. When enabled, the PMAlloc invokes the `getcpu()` system call periodically to check if the thread has migrated to another NUMA node. If such migration is detected, then the

items in the tcache are transferred back to the CPU-local arena, and the thread is bound to a new arena associated with its new CPU, thereby maintaining NUMA locality. This functionality can also be manually invoked through the `pmalloc_numa_migrate_check()` interface.

7 RECOVERY

The recovery code is invoked by `pmalloc_init()` when the allocator detects an unclean initialization, as evidenced by the presence of existing persistent heap files. 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 `pmalloc_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.

Normal shutdown recovery. 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. Within each arena, we scan all associated log files. Initially, we construct a *vchunk* header for each log chunk contained in these files. Subsequently, we examine the *alt* bit in each file's log header to retrieve the linked list of active log chunks. These active log chunks function as leaf nodes during the reconstruction of the red-black tree. The remaining free chunks in the log files are added to the arena's free list of log chunks. After that, 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), then 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.

Post-crash recovery. If a crash occurs during runtime or the recovery process, then we execute post-crash recovery. In this procedure, we first conduct the normal shutdown recovery to rebuild the DRAM metadata. Then, we additionally use different methods to do memory sanity check to resolve possible inconsistency issues according to the consistency model of allocators. For PMAlloc-LOG, we replay WALs to reverse the metadata changes of partially completed operations. We retain the old WALs until all log entries have been replayed to prevent the loss of log entries in the crash during recovery. For PMAlloc-GC, we conduct conservative garbage collection [8, 9] as in Makalu. This process is initiated from top-level root pointers with multiple garbage collection threads, which execute a parallel mark algorithm. The garbage collection threads conservatively scan the memory regions of root objects, treating any value within the address range of user data in the persistent memory heap as a potential pointer. Upon identifying a potential pointer, the corresponding target object is marked as reachable. We will then recursively identify pointers from this object. This process continues until no new objects can be marked, indicating the coverage of all allocated blocks. 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, then we undo all the operations of metadata transformation.

Completion of delayed remote deallocations. PMAlloc relies on deallocation logs and TSs to ensure crash consistency in the remote deallocation procedure. For a normal shutdown, PMAlloc triggers epoch-based synchronization for each arena, thereby ensuring the completion of all

outstanding remote deallocations. In the recovery phase, no additional work needs to be executed. However, if a system crash happens before the normal shutdown is completed, then PMAlloc needs to re-do log entries recorded in deallocation logs to recover to a consistent state. The re-do operations are executed after the post-crash recovery phase is completed. For each log entry, the re-do thread will compare its TS with the TS stored in the pslab whose persistent bitmap contains the corresponding bit of this log entry. If the TS of the log entry is older, it means the corresponding bit has been synchronized before the crash happens and this log entry can be skipped safely. If the TS of the log entry is newer, then the working thread will unset its corresponding bit in the regular bitmap and persistent bitmap to complete the release process of the recorded memory blocks. After all log entries have been checked, deallocation logs can be safely removed and PMAlloc is ready for serving new allocations and deallocations.

8 EVALUATION

8.1 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. We use the *numactl* utility to bind every thread to one core in the first socket to avoid the NUMA effects in all the experiments except those related to FPTree (Section 8.5) and NUMA effect (Section 8.4). In cases where the number of threads exceeds the available number of processor hyperthreads, additional threads are bound to CPU cores based on the modulus of their thread number against the total number of cores. All source codes are compiled with g++7.5 with -O3.

Compared allocators. We compare PMAlloc with state-of-the-art persistent allocators, including PMDK [17], nvm_malloc [71], PAllocator [63], Makalu [8], and Ralloc [10]. 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 article. We exclude jemalloc [26], Hoard [6], and tcmalloc [28], because they are volatile allocators. To support existing consistency models, we implement two versions of PMAlloc: *PMAlloc-LOG* and *PMAlloc-GC*, which leverage WAL and GC to keep crash consistency and avoid memory leaks, respectively. We choose the number of bit stripes as 6 in interleaved mapping, because it provides the optimal performance. The impact of the number of bit stripes is further discussed in Section 8.7.

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

8.2 Evaluations Using Benchmarks

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

Threadtest [6] 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 $s = 64$ B.

Prod-con [6, 70] 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 $s = 64$ B.

Shbench [60] is a stress test for an allocator. In each iteration, each thread allocates and frees objects of varying sizes from 64 B to 1,000 B. The smaller objects are allocated and freed more frequently. We run 10^5 iterations.

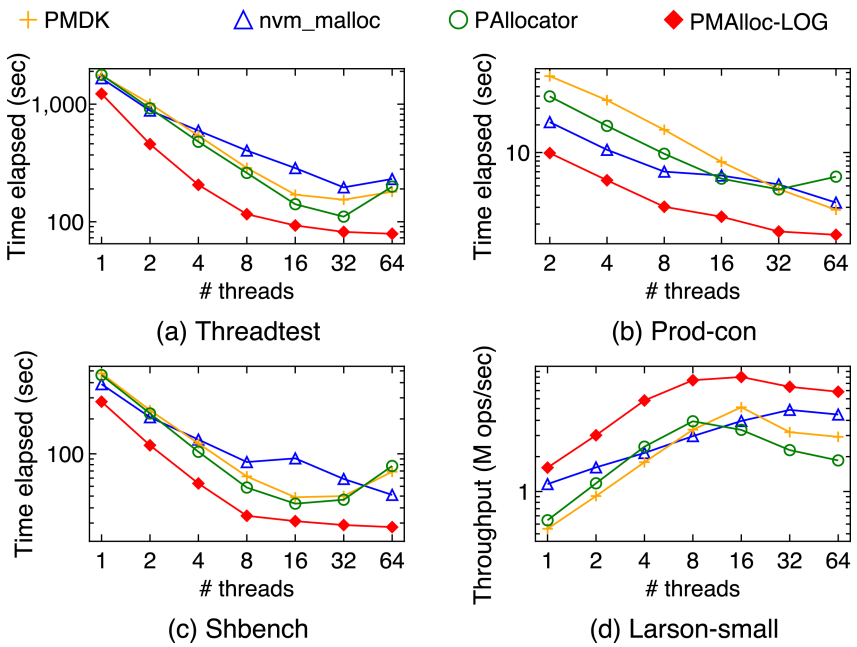


Fig. 16. Performance (log10 scaled) of small allocations with strongly consistent allocators.

Larson [49, 63] 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 [25]: it simulates the allocation in a database with TPC-DS benchmark for t threads. In each iteration, each thread 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 16 and Figure 18, respectively. Overall, PMAlloc outperforms and scales better than all the counterparts on all benchmarks.

Figure 16 shows that PMAlloc-LOG is up to 6.4×, 3.5×, and 3.9× faster than PMDK, nvm_malloc, and PAllocator, respectively, on the four benchmarks. PMAlloc-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 [22] to measure the breakdown of the execution time of different benchmarks with 8 threads. The execution time is normalized. The benchmark execution consists of object searching, splitting, and coalescing of extents in allocation/deallocation (denoted as Search), metadata flushing (FlushMeta), WAL flushing (FlushWAL), and others (Other). In the breakdown analysis, we fix the number of test operations at 10 million for the Larson benchmark, ensuring a fair comparison of time distribution. We conduct the experiment with five different versions of PMAlloc-LOG: *Base* denotes PMAlloc-LOG

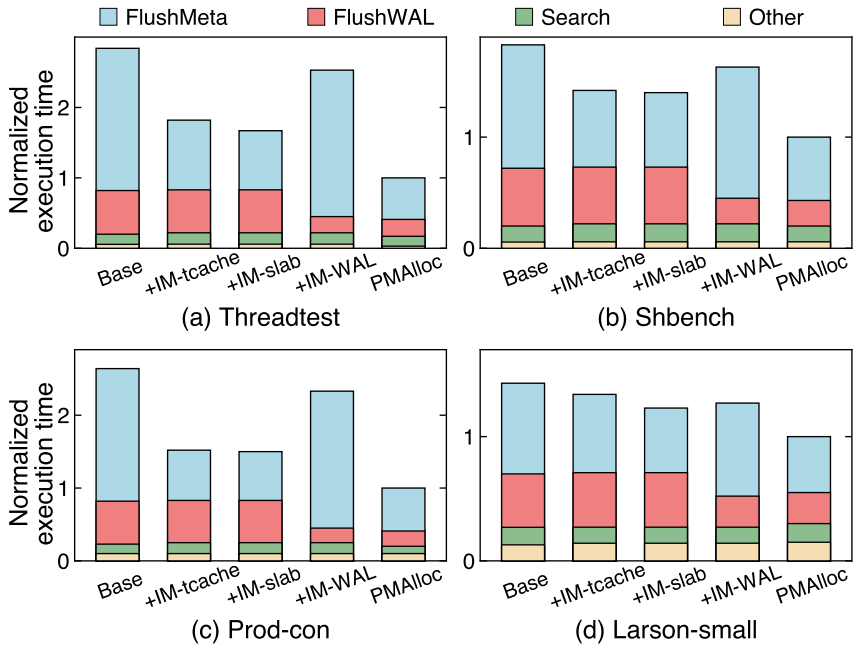


Fig. 17. Performance breakdown analysis of small allocations for LOG-based allocators.

without any optimizations described in Section 4. *+IM-tcache*, *+IM-slab*, and *+IM-WAL* correspond to versions where optimizations of interleaved tcache layout, interleaved mapping of slab bitmaps, or interleaved appending of WALs, are individually enabled. *PMAlloc* denotes the version where all the aforementioned optimizations are applied. As illustrated in Figure 17, the FlushMeta and FlushWAL time account for 51% to 87% of the execution time of *Base* across all benchmarks. Compared to *Base*, *+IM-tcache* reduces the FlushMeta time of *Base* by 14% to 62%, while *+IM-slab* diminishes it by 28% to 63%. With the interleaved appending of WALs, *+IM-WAL* achieves the reduction of 41% to 66% in FlushWAL time. The comprehensive optimizations of PMAlloc-LOG further reduce the total amount of flush time (FlushMeta and FlushWAL), resulting in an overall speedup ranging from 30% to 65%. Additionally, PMAlloc-LOG yields more benefits in Threadtest and Prod-con compared to the other benchmarks. This is because they have more cache line reflushes for their fixed allocation size.

Figure 18 shows that PMAlloc-GC achieves a maximal speedup of 70× and 6× over Makalu and Ralloc on the four benchmarks. PMAlloc-GC has better performance, because Makalu and Ralloc use the embedded linked list to manage free blocks in persistent slabs while PMAlloc-GC uses bitmaps to manage blocks, improving data access locality in persistent memory. PMAlloc-GC also maintains a volatile bitmap copy in DRAM for fast free block indexing and reducing accesses to persistent memory. To further illustrate the results above, we run the small allocation benchmarks with 32 threads and use Linux *perf* tools to collect L1-cache miss count at runtime. Note that we have fixed the number of operations at 10 million for Larson. Figure 19 shows that with PMAlloc-GC, all four benchmarks have the lowest number of L1-cache miss. And PMAlloc-GC reduces the miss count by a maximum of 70× and 2.8× compared to Makalu and Ralloc, proving the data locality enhancement achieved by bitmap utilization. To quantify the benefit of volatile bitmap copy, we run the four benchmarks using 32 threads. We compare PMAlloc-GC performance with the feature on and off. The normalized results are shown in Figure 20. It indicates that with volatile

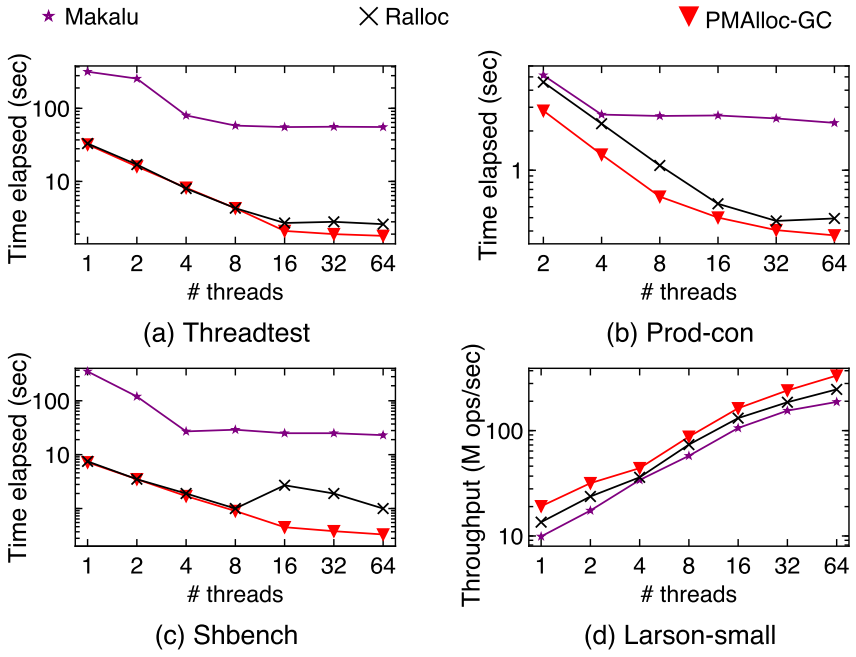


Fig. 18. Performance (log10 scaled) of small allocations with weakly consistent allocators.

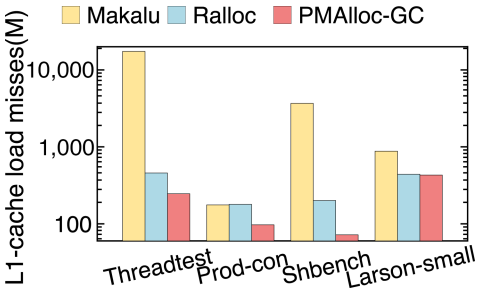


Fig. 19. L1-cache misses (log10 scaled) with weakly consistent allocators.

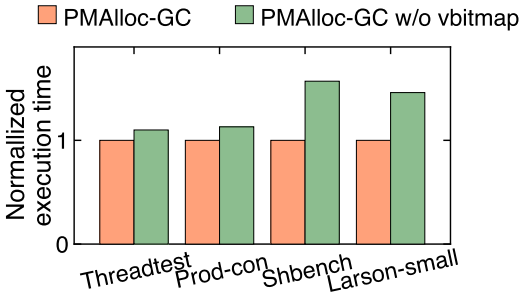


Fig. 20. Performance impact of the volatile bitmap copy. The *vbitmap* denotes volatile bitmap copy.

bitmap, PMAlloc-GC runs up to 1.57× faster than that with the volatile bitmap disabled for all the benchmarks.

Performance of large allocations. Figure 21 shows the performance of large object allocations. Because PMAlloc-GC performs the same as PMAlloc-LOG for large allocations, we exclude it in Figure 21. On Larson-large and DBMStest, PMAlloc-LOG is up to 40×, 18×, 55×, and 57× faster than PMDK, nvm_malloc, PAllocator, and Makalu. PMAlloc-LOG is faster than its counterparts because of using log-structured bookkeeping and the interleaved mapping in WALs.

To illustrate the impact of log-structured bookkeeping and interleaved mapping for large allocations, we evaluate the execution time breakdowns using these two benchmarks. Figure 22 shows the results. The *+Log* version signifies the implementation of log-structured bookkeeping exclusively. It reduces the total amount of flush time (FlushMeta and FlushWAL) by 28% and 45% in the two benchmarks, because the log-structured bookkeeping provides a sequential write pattern to

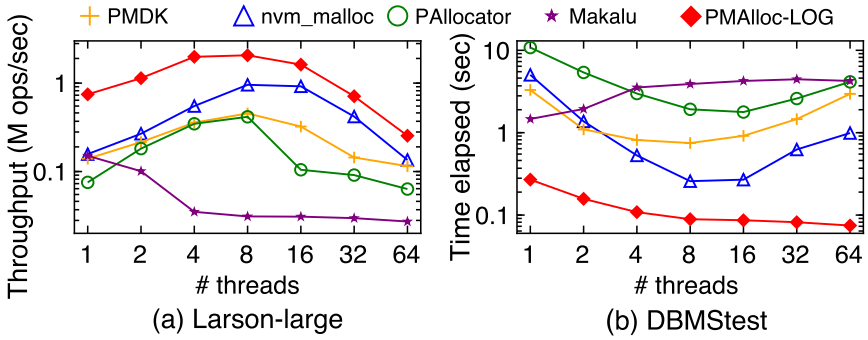


Fig. 21. Performance (log10 scaled) of large allocations.

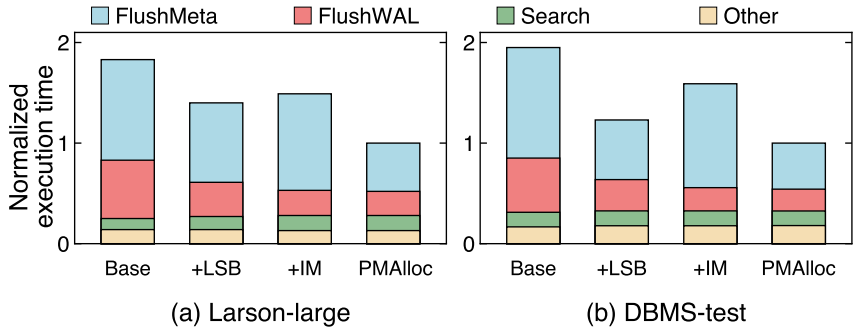


Fig. 22. Performance breakdown analysis of large allocations.

persistent memory. *+IM* exclusively enables interleaved mapping and gains a 23% speedup in flush time for both benchmarks due to eliminated repeated cache line flushes of log entries and WALs. The comprehensive optimizations integrated into PMAlloc-LOG achieve an overall speedup by up to 49%.

Space usage. Figure 23 and Figure 24 show the memory consumption of different allocators. Because the PMAlloc-LOG and PMAlloc-GC use the same amount of space, we only show the result of PMAlloc-LOG. PMAlloc-LOG's peak memory consumption is comparable to other allocators on all benchmarks. We exclude RAlloc in Figure 24, because RAlloc does not work correctly for large objects in their open-source implementation.

Impact of thread pinning. Considering that real-world applications may not employ thread-pinning techniques in their designs, we conducted supplementary tests using benchmarks without thread pinning to assess its impact. In this setup, threads are localized to the first NUMA node but are not explicitly bound to individual cores. The results are depicted in Figure 25 and Figure 26. PMAlloc continues to demonstrate superior performance compared to other allocators.

8.3 Evaluations Using Fragbench

We then evaluate PMAlloc on Fragbench [69] with the four workloads listed in Table 1 in Section 3. Figure 27(a) shows the space consumption of different allocators. We exclude the ones in Figure 5(b) except Makalu to avoid redundant representation. As PMAlloc-LOG and PMAlloc-GC yield the same space consumption, we only include PMAlloc-LOG. For comparison, we also evaluate

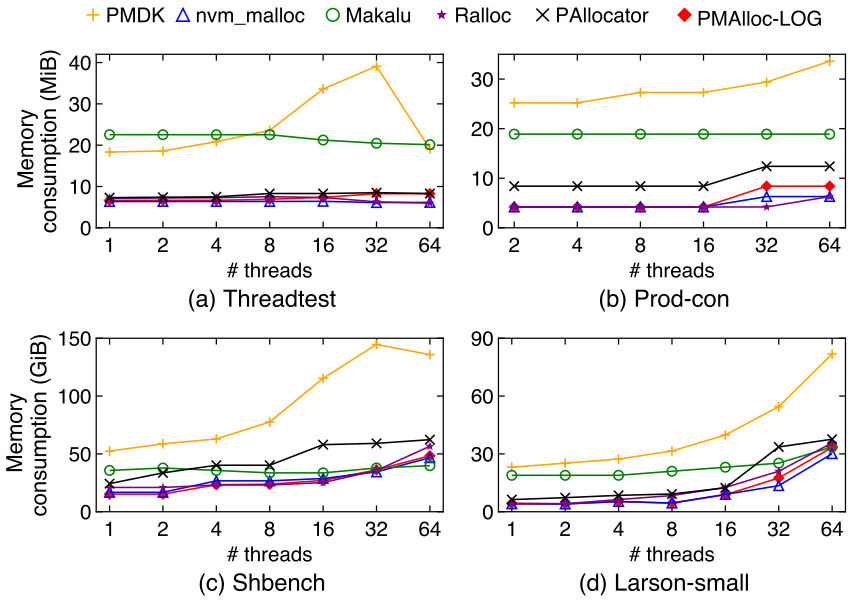


Fig. 23. Space consumption of small object allocations.

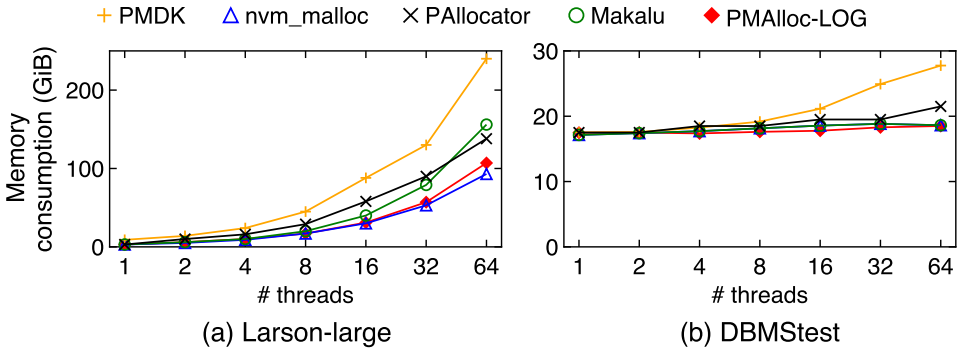


Fig. 24. Space consumption of large object allocations.

PMAlloc-LOG without the slab morphing strategy (PMAlloc-LOG w/o SM). The result shows that PMAlloc-LOG achieves the smallest space consumption because of the slab morphing technique.

To verify this, Figure 27(b) shows the space breakdown of PMAlloc-LOG. We divide the slabs into three categories according to their memory utilization: 0%–30%, 30%–70%, 70%–100%. Figure 27(b) shows that, with the slab morphing, PMAlloc-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. The slab morphing has more utilization improvement with W1 and W3, because they both have a 90% deletes in *Delete phase*, causing more slabs to become candidates of slab morphing. For W2 and W4, they have fewer delete operations (0% and 50% in *Delete phase*), making many slabs still have a high memory utilization after the *Delete* and *After* phases and can not be selected as morphing candidates. A larger SU can let more slabs be morphed with a higher performance overhead. We discuss this in Section 8.7.

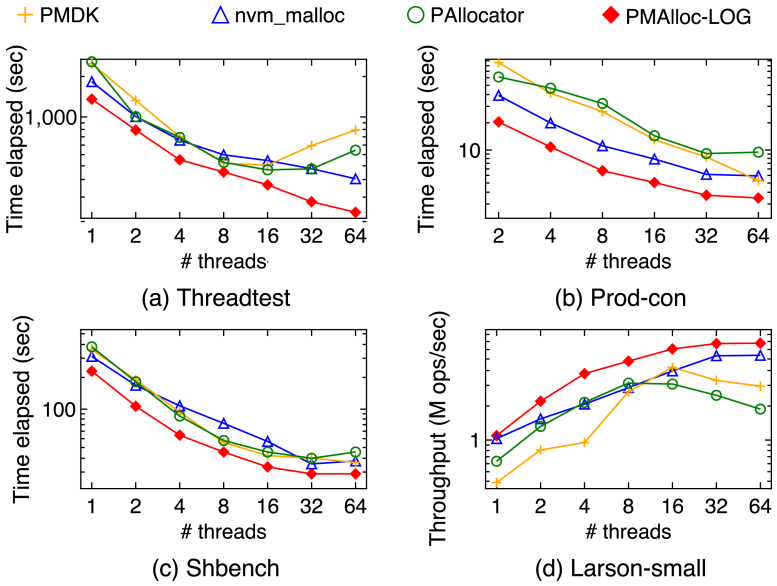


Fig. 25. Performance (log10 scaled) of strongly consistent allocator without thread binding.

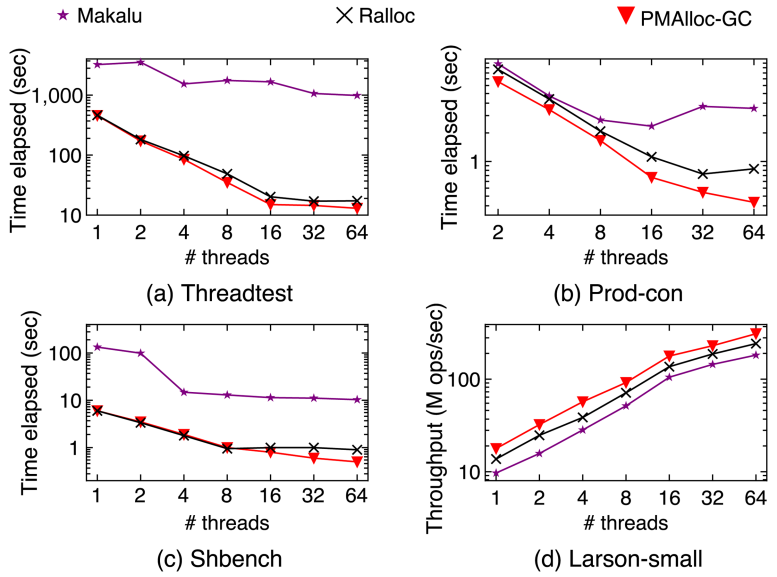
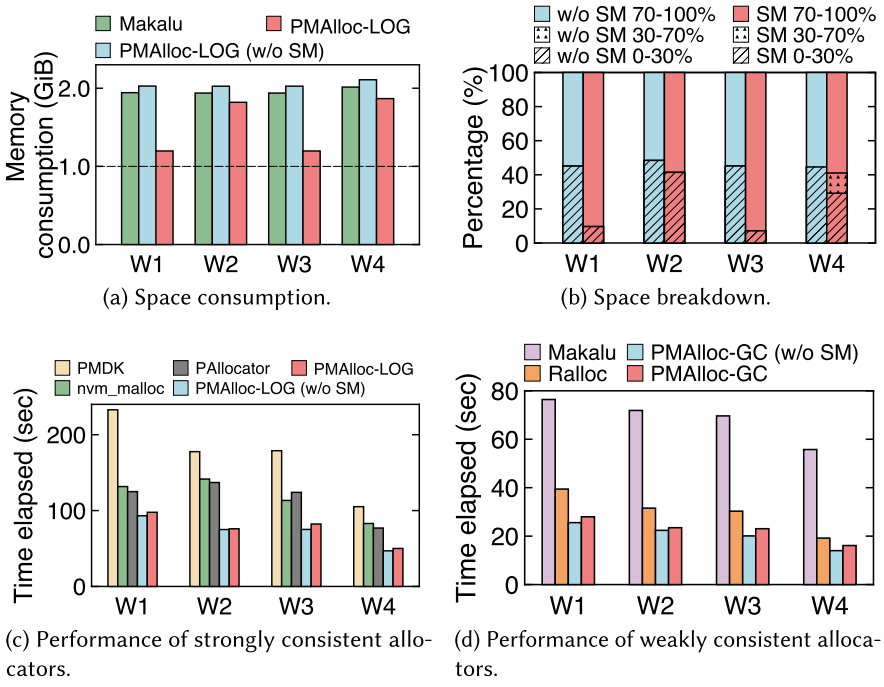


Fig. 26. Performance (log10 scaled) of weakly consistent allocator without thread binding.

Figures 27(c) and 27(d) show the performance of PMAlloc. PMAlloc outperforms all other allocators because of using the interleaved mapping technique, as discussed in Section 8.2. 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%.



(a) Space consumption.

(b) Space breakdown.

(c) Performance of strongly consistent allocators.

(d) Performance of weakly consistent allocators.

Fig. 27. Results of Fragbench. SM denotes slab morphing.

To substantiate the efficacy of slab morphing, we monitored the quantity of slabs in three distinct states (i.e., $slab_{before}$, $slab_{in}$, and $slab_{after}$) throughout the execution of Fragbench. We modify Fragbech by expanding the size of the total allocated memory of the After phase from 5 GB to 10 GB to further show the transform ratio of $slab_{after}$ in a longer running time. The results are presented in Figure 28. We denote the original After phase as *After1* and the expanding part as *After2*. We observe that slab morphing primarily occurs in the After phase. This is because (1) most allocation requests that arrived in the *Before* phase belong to the same size classes, and (2) no allocation happens in the *Delete* phase. Conversely, during the After phase, slabs undergo deallocations from the *Delete* phase and receive new allocation requests of different size classes. Figure 28 shows that for W1 and W3, PMAlloc morphs 37.2% and 43.6% of slabs into $slab_{in}$, respectively, followed by 57.9% and 48.8% of these $slab_{in}$ completing the morphing process to transition into $slab_{after}$ by the end of the *After1* phase. This is because the morphing mechanism obviates the need for new slab allocations, reutilizing existing slabs and thus significantly reducing memory consumption. For W2 and W4, PMAlloc transformed 14.4% and 22.2% of $slab_{before}$ into $slab_{in}$, respectively, and eventually 11.6% and 17.2% of $slab_{in}$ complete the morphing. Additionally, during *After2* phase, PMAlloc keeps morphing slabs to serve new allocation requests, and eventually most of $slab_{in}$ is transformed into $slab_{after}$. By the end of the four workloads, 98.3%, 96.1%, 97.2%, and 95.7% of $slab_{in}$ complete the transformation process.

8.4 Effectiveness of NUMA-aware Allocations and Deallocations

We use all the four benchmarks to show the performance of PMAlloc with NUMA cores. In each benchmark, we increase the number of threads from 16 to 80. As the prior work did [10], we first pin each thread to one core on the first NUMA node and then to one hyperthread on the same node. When the number of threads exceeds the maximum number of hyperthreads on one node (i.e., 40),

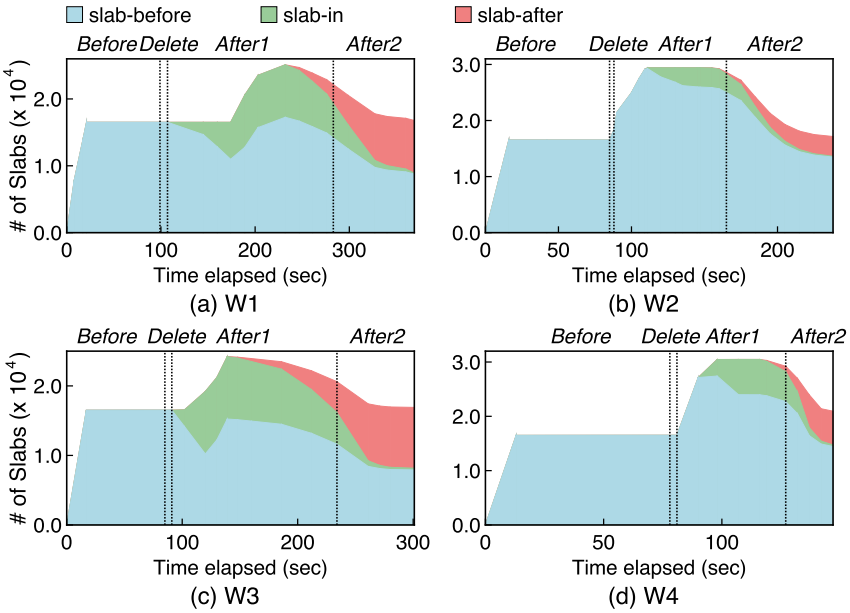


Fig. 28. Slab state statistic during Fragbench execution.

we pin the remaining threads to the other node. In the Prod-con test, we still pin the producer threads to the first node and the consumer threads to the second node, as described in Section 3.5. For PMDK, we leverage its pool set feature to unify persistent memory heaps across two nodes. Other persistent memory allocators lacking multi-heap capability allocate memory blocks only on the first NUMA node and directly release remote blocks. To further highlight the benefits for using our NUMA optimizations, we add a compared system, named PMAllocNN, which denotes PMAlloc without enabling NUMA optimizations. PMAllocNN has similar allocation and deallocation behaviors to existing allocators but enables metadata management optimizations proposed in Section 5.

Figure 29 shows the results of strongly consistent allocators. PMAlloc-LOG achieves up to 2.9× performance improvement over PMAllocNN-LOG and 7.3× performance improvement over other allocators. For Threadtest, Shbench, and Larson, the performance improvement is mainly from our NUMA-aware allocation, because all of these benchmarks have almost no cross-thread releases (< 1%) [2]. When the number of threads exceeds 40, these benchmarks will run on two nodes. The performance improvement is because PMAlloc-LOG can allocate memory blocks on the local node of the working threads, while other allocators are unaware of the NUMA effect, resulting in allocations on the remote node. For the Prod-con benchmark, allocators always allocate local memory blocks but release them remotely. PMAlloc-LOG achieves up to 4.3× performance improvement over other allocators and 1.4× over PMAllocNN-LOG. This is because the two-phase NUMA-aware deallocation of PMAlloc-LOG eliminates remote NUMA accesses in the remote release procedure.

To further explore the reason for the performance improvement, we conduct a persistent memory traffic breakdown analysis of PMAlloc-LOG and PMAllocNN-LOG on all four benchmarks with 80 threads. We use Intel's IPMCTL tools [34] to record the total amount of data accessing persistent memory. We then use Intel's PCM tools [35] to record the amount of data accessed from remote NUMA nodes as Nap [74] did. Figure 30 shows the results. PMAlloc-LOG reaches nearly

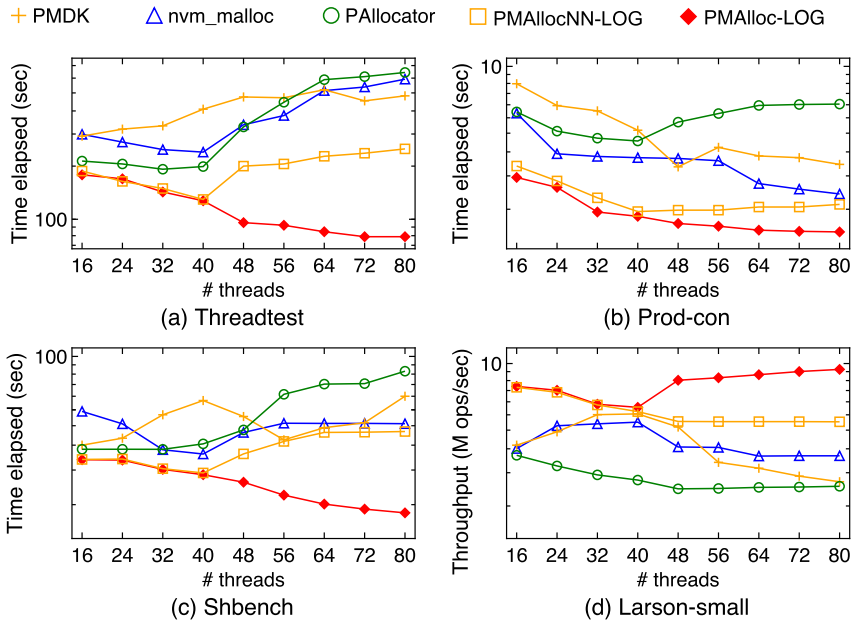


Fig. 29. NUMA test for strongly consistent allocators.

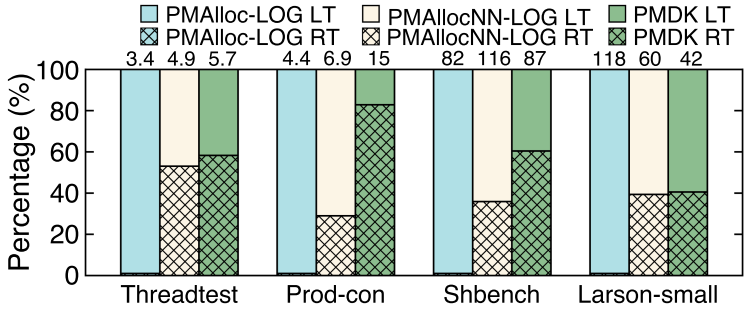


Fig. 30. Breakdown analysis of persistent memory traffic. The absolute number (GB) of total traffic is marked on the top. LT is short for local traffic and RT is short for remote traffic.

Table 2. The Amount of Data Written to Internal Media of Persistent Memory

Benchmark	PMAlloc	PMAllocNN	PMDK
Threadtest	0.5 GiB	2.1 GiB	6.8 GiB
Prod-con	2.1 GiB	4.9 GiB	11 GiB
Shbench	45 GiB	75 GiB	99 GiB
Larson-small	135 GiB	62 GiB	62 GiB

zero remote traffic in all four workloads, while the baseline allocators issue remote accesses that account for 29% to 82% of all the persistent memory accesses.

We also measure the total amount of data written to the internal media of persistent memory for PMAlloc-LOG, PMAllocNN-LOG, and PMDK with IPMCTL tools. Table 2 shows the results. PMAlloc-LOG reduces persistent memory writes by up to 76% and outperforms PMDK by up to

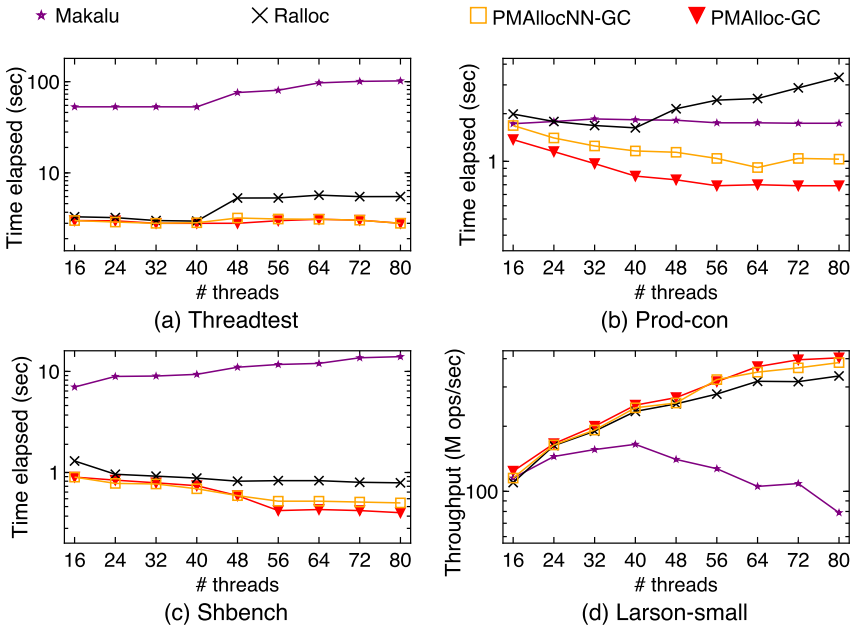


Fig. 31. NUMA test for weakly consistent allocators.

92% in Threadtest, Prod-con, and Shbench. Larson has a fixed amount of execution time, while other benchmarks have fixed amount of operations. It writes 2.2× more data with PMAlloc-LOG than that with baseline allocators, because the former improves its throughput by 2.9×. The main reason for such write reduction is PMAlloc-LOG reduces remote NUMA accesses, which induces extra writes to the persistent media for maintaining cache coherency under directory coherence protocol.

We also evaluate the PMAlloc-GC with weakly consistent allocators. Figure 31 shows that PMAlloc-GC achieves up to 36× performance improvement over other allocators and 1.45× over PMAllocNN-GC. The improvement ratio of PMAlloc-GC compared to PMAllocNN-GC with weakly consistency model is smaller than that with the strongly consistency model. This is because most cache line flush operations are eliminated in weakly consistency model so the cache lines from the remote nodes can be retained in the local processor cache. This will hide the high latency of subsequent memory accesses.

8.5 Evaluation Using FPTree

We also evaluate PMAlloc with a real-world key-value store application, FPTree [64], under two NUMA nodes. 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 [11], 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 32 shows the throughputs of FPTree using different allocators. With PMAlloc-LOG, FPTree yields up to 2.5×, 2.7×, and 3.1× throughput compared with PMDK, nvm_malloc, and

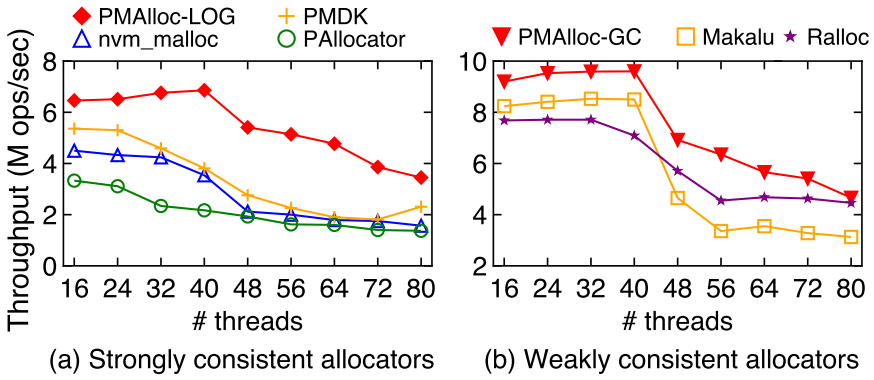


Fig. 32. Performance of FPTree with two NUMA nodes.

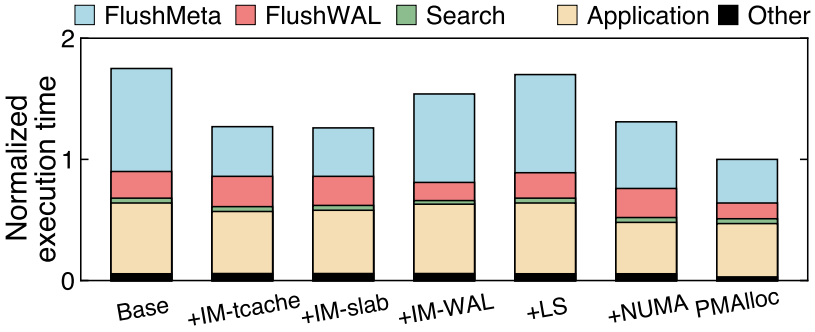


Fig. 33. Performance breakdown of FPTree.

PAllocator, respectively. With PMAlloc-GC, FPTree brings a speedup up to 1.6×. FPTree with PMAlloc yields comparable space consumption over other allocators, since the slab morphing technique is not triggered for the given workload.

We further show the execution time breakdown of FPTree to study the impact of individual optimizations in PMAlloc-LOG. As in Section 8.2, we use the Linux *perf* tool to measure the execution time. Figure 33 displays the results. We introduce a new category to represent the time spent on executing the code that is not related to allocators. We denote it as Application. The *+NUMA* version indicates a configuration where only NUMA-related optimizations are activated. Compared to *Base*, *+IM-tcache* decreases FlushMeta by 51.8%, while *+IM-slab* reduces it by 53.1%. Employing interleaved appending of WALs, *+IM-WAL* cuts FlushWAL by 31.9%. The *+LS* version reveals that log-structured bookkeeping only trims the execution time by 2.7%. This marginal impact is attributed to the infrequency of large allocations in this application. *+NUMA* further reduces the total execution time by 26.4%. This is because the NUMA optimization uses more local memory to serve requests and eliminates the remote accesses when executing remote deallocations. Overall, *PMAlloc* with all optimizations can reduce the execution time by 43% compared to *Base*.

8.6 Evaluation Using Real-world Applications

To further demonstrate PMAlloc's performance and scalability, we evaluate it using real-world applications. We use two NUMA nodes in the experiments. These applications are adapted from

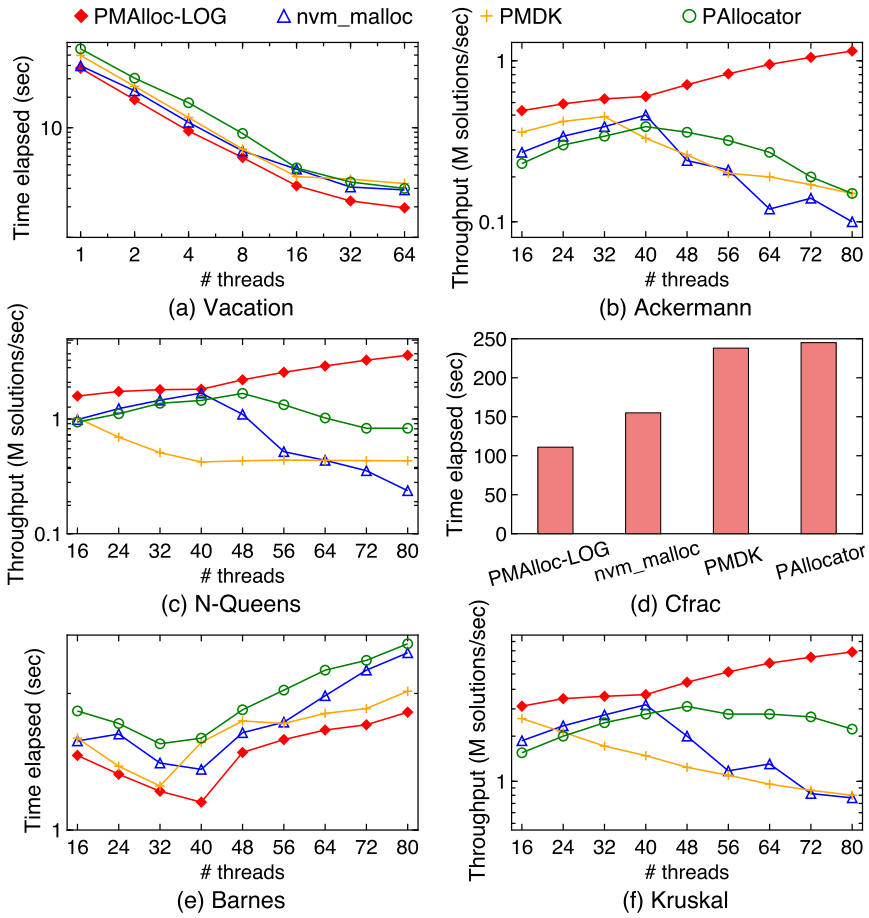


Fig. 34. Performance of real-world applications for strongly consistent allocators.

various use cases. We replace their default allocators with persistent memory allocators and then run all the applications with varying numbers of threads.

Vacation [61, 72] is an online transaction processing system, which uses a red-black tree in its internal database. We configure the database to contain 16,384 possible relations and conduct 1,000,000 transactions. Each transaction consists of five queries that target approximately 90% of these relations. Our setup aligns with that of Ralloc [10]. Note that Vacation requires that the number of threads must be a power of 2. Figure 34(a) shows that PMAlloc-LOG outperforms PMDK, nvm_malloc, and PAllocator by up to 1.7×, 1.4×, and 1.5×, respectively. And Figure 35(a) shows that PMAlloc-GC performs up to 2.9× and 1.2× better than Makalu and Ralloc.

Ackermann [1] is a recursive implementation for a mathematical function named after Wilhelm Ackermann. It performs six 600-byte allocations as caches to record the parameters and results of each recursive step. Similar to Poseidon [23], we configure this application to repeatedly calculate a set of Ackermann functions 10 million times. The performance of different allocators is shown in Figure 34(b) and Figure 35(b). For the strongly consistent allocators, it shows that PMAlloc-LOG achieves 7.7×, 11.5×, and 7.6× higher throughput compared to PMDK, nvm_malloc, and PAllocator, respectively. Due to the NUMA-aware allocation and deallocation design of

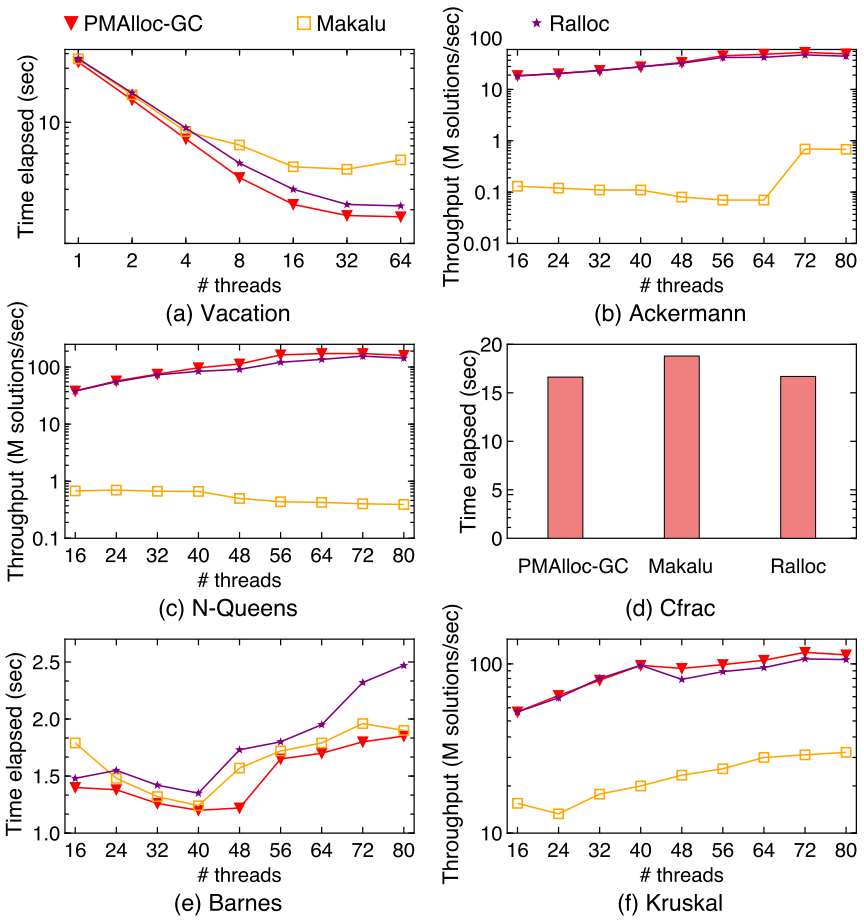


Fig. 35. Performance of real-world applications for weakly consistent allocators.

PMAlloc, the throughput with PMAlloc-LOG is increased as the number of threads is increased while the other allocators struggle in performance. For the weakly consistent allocators, PMAlloc-GC shows similar performance to Ralloc but delivers a 39× higher throughput than Makalu.

N-Queens [7] is a multi-threaded implementation designed to solve the n-queens problem. The application solves n-queens puzzles on an 8×8 board, utilizing a 128-byte allocation, which is deallocated upon completion of the puzzles. We repeatedly find the solution for the 8-queens puzzle 100,000 times, using a variable number of threads. Figures 34(c) and 35(c) show the results. PMAlloc demonstrates superior scalability compared to other allocators, particularly when the thread count surpasses 40 and threads are distributed across different NUMA nodes. With 80 threads, PMAlloc-LOG outperforms PMDK, nvm_malloc, and PAllocator by 8.3×, 15.1×, and 4.3×, respectively. PMAlloc-GC also achieves 278× and 1.26× better throughput compared with Makalu and Ralloc.

Cfrac [52] is an implementation of the continued fraction factoring algorithm. It is a single-threaded test that involves many short-lived, small memory allocations. We run this application and factor 44-digit number 17,545,186,520,507,317,056,371,138,836,327,483,792,789,528, which is the product of two primes. Figure 34(d) shows that PMAlloc-LOG reduces the execution time of

Cfrac by 53%, 28%, and 55% compared with PMDK, nvm_malloc, and PAllocator. Figure 35(d) indicates that Cfrac performs similarly with both Ralloc and PMAlloc-GC but is 11% faster than Makalu.

Barnes [5] is a multi-threaded algorithm for solving the gravitational N-body problem. This application performs relatively fewer allocations than Cfrac. It is widely used by many volatile and non-volatile memory allocators [6, 8, 53]. We set the number of particles being simulated as 400,000. Figure 34(e) shows that PMAlloc performs similarly with PMDK and outperforms nvm_malloc and PAllocator by 17% and 21%, respectively. From Figure 35(e), we observe that there is minimal performance difference between PMAlloc and Ralloc, while Makalu exhibits a performance decline of 22% compared to them.

Kruskal [47] is a graph processing algorithm. It aims to find the **minimal spanning tree (MST)** in the graph. Similar to Poseidon, we configure the application to solve Kruskal MST implementations for graphs of order 6. Each execution involves allocating a 168-byte block for temporary graph data, employing a greedy strategy for problem-solving, and then deallocating the memory. This process is repeated 10 million times. Figure 34(f) shows that, unlike other memory allocators, PMAlloc continues to exhibit performance improvement after the number of running threads exceed 40 and yields up to 7.6×, 7.9×, and 3.1× throughput compared with PMDK, nvm_malloc, and PAllocator. For weakly consistent allocators, we can observe from Figure 35(f) that PMAlloc-GC exhibits comparable performance to Ralloc while achieving a maximum throughput increase of 3.2× over Makalu.

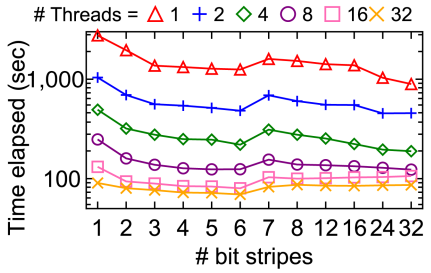
8.7 Sensitivity Analysis

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 [79] in persistent memory when a large number of cache lines flush concurrently. To explore the impact of the number of bit stripes, we run PMAlloc-LOG on Threadtest with varying numbers of threads as a study case.

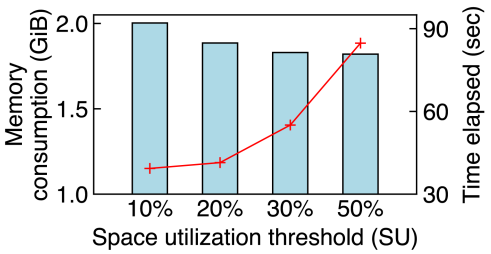
As Figure 36(a) shows, the execution time of PMAlloc-LOG is not linearly decreased as we increase the number of bit stripes. This is because the execution time is determined by both software parameters (i.e., the number of bit stripes and the number of threads) and hardware parameters (i.e., the number of XPBuffer lines in persistent memory and its size). In this article, we choose the number of bit stripes as 6, because it achieves the best performance for most cases. Users may adjust the bit stripe size to better suit the requirements of specific applications.

Morphing parameter. The slab **space utilization threshold (SU)** in the morphing technique also impacts the efficiency of PMAlloc. 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 36(b) shows the impact of SU on PMAlloc-LOG on the W4 workload. Based on the results, we empirically set SU as 20% to achieve a decent tradeoff 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.

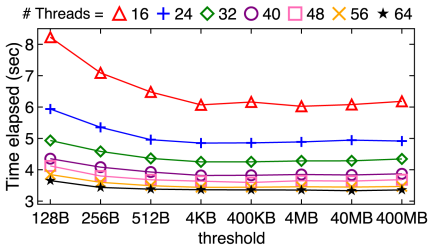
Deallocation log size threshold. In the NUMA-aware two-phase deallocation mechanism, the deallocation log size threshold defines the maximum capacity of the deallocation logs. A larger threshold results in a higher occupation of memory by these logs, while a smaller threshold triggers epoch-based synchronization more frequently, potentially impacting overall performance. To investigate the influence of the deallocation log size threshold on allocator performance, we run PMAlloc-LOG on the Prod-con benchmark with a varying number of threads. As the deallocation log size threshold increases, Figure 36(c) shows that the allocator's performance initially



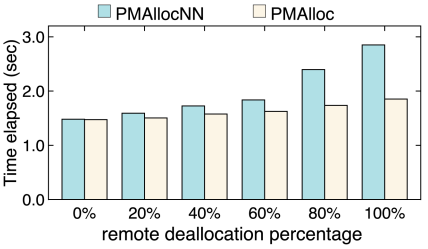
(a) Impact of the number of bit stripes on Threadtest.



(b) Impact of the morphing parameter on Fragbench.



(c) Impact of deallocation log size threshold on Prod-con.



(d) Impact of remote deallocation ratio on Prod-con.

Fig. 36. Sensitivity analysis.

deteriorates with a very small threshold but stabilizes when the threshold reaches 400 KB or larger. To strike a balance between memory occupation and performance, we set the deallocation log size threshold to 4 MB in our experimental configuration.

Remote deallocation ratios in applications. Different applications may consist of varying ratios of remote deallocations, ranging from 0% to 100%. To evaluate the impact of our two-phase remote deallocation technique on different applications, we conducted tests using the Prod-con benchmark. Specifically, we varied the percentage of remote deallocations in Prod-con, ranging from 0% to 100%. We run it using 40 threads for both PMAlloc and PMAlloc-NN (with NUMA optimizations disabled). The results are shown in Figure 36(d). Both allocators exhibit similar performance when the ratio of remote deallocations is 0%. However, as this ratio increases, PMAlloc outperforms PMAlloc-NN with a performance gain ranging from 1.06× to 1.54×. These results indicate that PMAlloc offers substantial benefits when an application has a high ratio of remote deallocations and remains effective at lower ratios.

8.8 Overhead Discussion

GC overhead. To evaluate the efficiency of log cleaning on log-structured bookkeeping for large allocations, we run PMAlloc-LOG on Larson-large and DBMStest. Figure 37 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 lightweight, since it only keeps the allocation metadata, thus the copying overhead is low.

Furthermore, Figure 38 shows the log file size during the benchmark execution. For Larson-large, we set the testing iterations to 10^4. To better illustrate the impact of GC operations, we increase the testing iterations to 500 for DBMStest. We can find that enabling GC significantly reduces the memory overhead. The frequency of GC execution can be varied. In Larson-large, PMAlloc

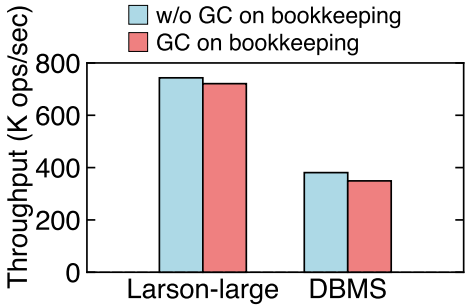


Fig. 37. GC overhead.

Table 3. Recovery Time

Allocators	10M	50M	100M
nvm_malloc	0.32 ms	1.3 ms	5.6 ms
PMDK	34 ms	54 ms	127 ms
PMAlloc-LOG	45 ms	97 ms	173 ms
Ralloc	0.55 s	2.1 s	6.7 s
Makalu	0.91 s	8.3 s	23.5 s
PMAlloc-GC	0.93 s	4.9 s	12.4 s

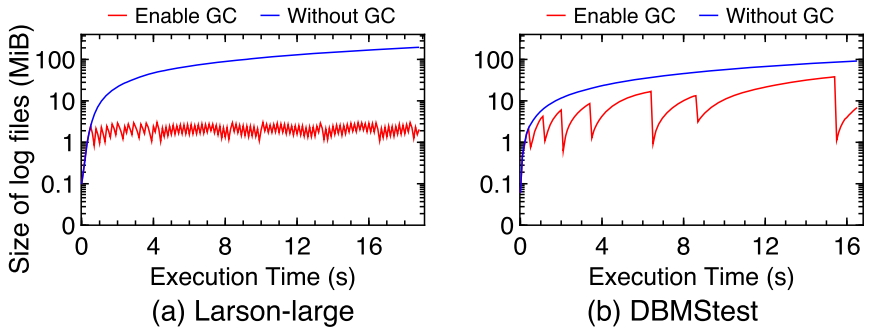


Fig. 38. Log file size (log10 scaled) of large allocations.

triggers 98 fast GCs and 98 slow GCs. In contrast, for DBMStest, it triggers 38 fast GCs and 7 slow GCs.

Recovery. Table 3 presents the recovery times of various open-source allocators. These allocators are tested with a linked list containing 10 million (10M), 50 million (50M), and 100 million (100M) nodes, each allocated with node sizes uniformly distributed between 64 bytes and 128 bytes. The recovery operations are performed using a single thread.

For strongly consistent allocators, PMAlloc-LOG is slower than PMDK and nvm_malloc across all tested sizes. This is because PMAlloc-LOG needs to scan both the WALs and log-structured bookkeeping, while PMDK only traverses the WALs and nvm_malloc defers some metadata reconstruction to the runtime deallocation process. The recovery time for PMAlloc-LOG correlates with the heap size, as the log-structured bookkeeping accounts for 0.2% of it. Thus, with more allocated nodes, the recovery time increases. However, since the log-structured bookkeeping is tailored for large memory objects, whose number is far fewer than smaller ones, the recovery time of PMAlloc-LOG remains acceptable (173 ms for 100 million memory objects) and is significantly reduced compared to GC-based allocators.

For weakly consistent allocators, their recovery process involves reconstructing DRAM metadata for both small and large memory objects by scanning the entire heap, resulting in extended recovery times. PMAlloc-GC shows comparable performance to Makalu. It is slower than Ralloc, because Ralloc only needs to scan part of nodes in the recovery.

Synchronization overhead. Background synchronization threads perform epoch-based synchronization periodically in NUMA-aware deallocations, potentially leading to contention with working threads. To assess the influence of possible thread contention, we run PMAlloc-LOG

Table 4. Synchronization Overhead of Deallocation Logs

Benchmarks	workload thread active time ratio (%)	synchronization thread active time ratio (%)
Threadtest	99.90	1.56×10^{-3}
Prod-con	98.33	1.326
Shbench	99.99	1.10×10^{-4}
Larson-small	99.95	4.66×10^{-5}

across four benchmarks with 80 workload threads. We employ *clock_gettime()* functions to measure the active duration of each thread using system clocks. By summing the active time of both workload threads and synchronization threads, we obtain their ratio in the entire process. As illustrated in Table 4, the overhead imposed by synchronization threads has a negligible impact on system utilization. Across Threadtest, Shbench, and Larson benchmarks, which have almost no cross-thread deallocations, the synchronization overhead is nearly zero. In the case of Prod-con, where all blocks are deallocated remotely, the synchronization overhead is more noticeable but remains remarkably low, accounting for less than 2% of the total active time. This is because the majority of remotely deallocated blocks (83% in our experiment) have been synchronized by the allocating thread incidentally through on-demand synchronization. The remaining blocks handled by background threads are synchronized in a NUMA-local manner, which involves only the slab headers within the local node, leading to minimal CPU contention with the application threads.

8.9 Evaluation on Emulated eADR Platform

eADR (extended ADR) is a new feature supported in the third-generation Intel Xeon Scalable Processors, which ensures CPU caches are also in the power-fail-protected domain [18]. 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 [68]. In this section, we evaluate PMAlloc 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 and have the same performance numbers as ADR ones.

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 39, the number of bit stripes has no impact on the performance of PMAlloc-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 $pmem\_has\_auto\_flush()$ in PMDK [17] 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 40 show that PMAlloc-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 PMAlloc-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 41 shows the performance of PMAlloc-LOG for large allocations. We can observe that it has an 11× performance improvement on average with Larson-large and DBMStest. This is

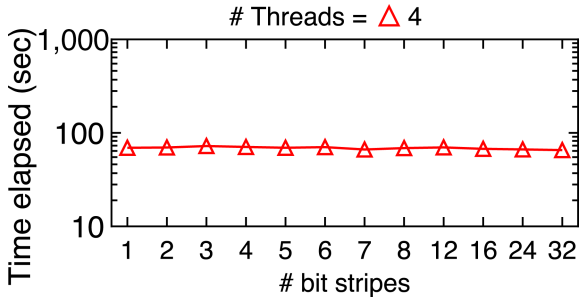


Fig. 39. Impact of interleaved mapping on eADR.

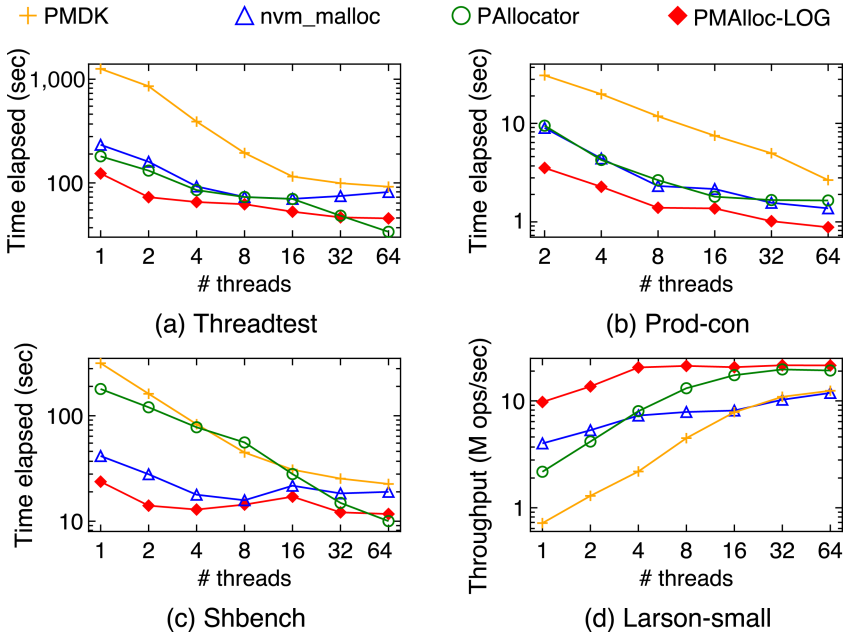


Fig. 40. Performance (log10 scaled) of small allocations on the emulated eADR platform.

because our design of VEH and log-structured bookkeeping reduces the total number of persistent memory access and improves the write locality for eADR.

9 DISCUSSION

In this section, we discuss the applicability of the proposed techniques to future persistent memory systems.

Cache line reflushing issue. The challenge of cache line reflushing originates from the CPU's instruction architecture design rather than the persistent memory hardware. In processors such as Intel's Cascade Lake and Skylake, a repeated flush instruction can only execute after previous flushes have completed and the same cache line is reloaded back, leading to long reflush latency [13, 78]. Newer processors, such as the recently released Ice Lake, have fully implemented the clwb instruction. In earlier processors, clwb was equated with clflushopt [18]. The clwb instruction can keep the cache line content in the processor cache during flushing, thereby mitigating cache line

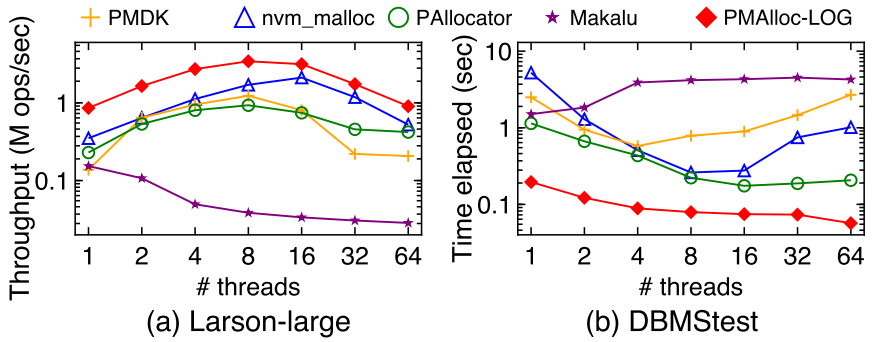


Fig. 41. Performance (log10 scaled) of large allocations on the emulated eADR platform.

reflush issues for waiting for data reloading. However, only the local clwb (pertaining to the same NUMA node) possesses this feature. For remote clwb and other instructions, such as clflushopt and clflush, the cache line content is not retained [18]. Thus, high reflush latency remains a significant performance hurdle. Our interleaved mapping technique provides a solution for these cache line reflush problems.

Random accesses impact. Random accesses can degrade the performance of persistent memory applications. Intel Optane DIMMs, for example, use an on-chip buffer to consolidate adjacent data requests, minimizing slow access times to the internal 3D-XPoint media. However, if the data request addresses are randomized, then this buffer loses its effectiveness, resulting in a performance decline. While Intel ceased production of new Optane DIMMs in the summer of 2022 for commercial reasons, we believe our log-structured bookkeeping technique remains relevant for upcoming byte-addressable persistent memory products. Devices like CXL-based products (e.g., Samsung's Memory-Semantic SSD [30, 42]), which are seen as potential replacements for Optane DIMMs, still employ an on-chip buffer for consolidating data requests. Consequently, when allocating large memory extents on such devices, the log-structured bookkeeping technique can help counteract the negative effects of small, random accesses caused by allocators.

Fragmentation concerns. Fragmentation due to static slab segregation is a prevalent problem in both volatile and persistent memory systems. Our innovative slab morphing technique can substantially cut down memory usage in applications with fluctuating allocation patterns, irrespective of the memory media in use.

NUMA impact. Recent studies [80] indicate that future persistent memory systems designed using the NUMA architecture might continue facing extended memory latencies due to remote memory access when applying the directory coherence protocol in Intel X86 machines. Hence, our suggested NUMA-aware memory allocation and deallocation techniques retain their relevance, ensuring efficient use of persistent memory in such environments.

10 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 [71] 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 [63] 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 [23] is the first persistent memory allocator enforcing page-based protections. It also uses bitmaps and logs for heap metadata management.

GC-based allocators. To reduce the overhead of writing logs and flushing metadata, recent allocators [8, 10, 59] employ **garbage collection (GC)** to reconstruct heap metadata after a crash. This process involves traversing the persistent heap and utilizing pointer identification techniques to locate and recover still-active memory blocks from its pre-defined root pointers. For instance, Makalu [8] pioneers this approach by utilizing offline GC to ease heap metadata persistence constraints when online. This results in accelerated small-block allocations. Similarly, Ralloc [10] changes the transient, lock-free allocator LRalloc into a persistent one, and like Makalu, Ralloc relies on post-crash GC to avert cache line reflushes. However, DCMM [59] 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 [17]. 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 PMAlloc 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, PMAlloc no longer has the memory fragmentation issue caused by static slab segregation.

NUMA-aware optimizations for volatile memory system. To avoid the penalty brought by remote NUMA access, volatile memory allocators have adopted NUMA-aware memory management policies. Kaminski et al. [43] make the working threads always allocate from the local memory first. Wagle et al. [73] design the allocator to always use the closest NUMA node to allocate memory when local memory runs out. TintMalloc [65] allows users to determine which NUMA node to make memory allocations and ensures the memory isolation between different tasks. However, these works only achieve NUMA locality in the allocation process but can not reduce the remote memory access induced by remote releases. PMAlloc addresses both issues while ensuring crash consistency on persistent memory. There are also numerous works [21, 24, 27, 55, 66] that optimize NUMA memory accesses through kernel-level page scheduling techniques. Carrefour [21] utilizes page replication as a means to distribute access pressure and reduce remote memory accesses. AsymSched [55] periodically samples memory access metrics and dynamically migrates pages between nodes to optimize performance. kMAF [24] optimizes memory access patterns by leveraging page fault tracing for efficient thread and data mapping. Unlike these kernel-level works, PMAlloc and other memory allocators aim to address remote access issues at the user level.

NUMA-aware optimizations for persistent memory system. Because persistent memory is more sensitive to NUMA impact than DRAM, recent works have tried to reduce remote memory access. Nap [74] provides a black-box approach to reduce remote access to persistent memory for index structures. It utilizes a global and volatile view in DRAM to cache hot items and absorb remote memory access. ListDB [45] mitigates NUMA impact in the skip list by making the upper layer pointers point only to the skip list elements on the same NUMA node. PACTree [46] finds that the root cause of the limited cross-socket bandwidth is the directory coherence protocol used in the x86 machines. ODINFS [80] uses local threads to delegate remote memory access requests, thereby eliminating remote NUMA access in the file system. POSEIDON [23] is a persistent memory allocator that considers the NUMA impact through its use of per-CPU sub-heaps, which naturally facilitate NUMA-local memory allocations. PMAlloc enhances POSEIDON's allocation strategy by permitting memory sharing across different NUMA domains when the local PM is

depleted. Moreover, PMAlloc is the first persistent memory allocator to consider mitigating remote metadata access during the deallocation operations.

Asynchronous deallocation. Asynchronous deallocation is often used to improve system performance by scheduling it at a later time using different mechanisms. For example, LATR [48] decreases the latency of *munmap()* by delaying expensive TLB shootdowns. It introduces a lazy software-based TLB shootdown mechanism in the kernel space that asynchronously invalidates TLB entries on context switches. DaxVM [3] proposes an asynchronous unmapping approach for DAX-based persistent memory files. It tracks the pages to unmap and defers their unmapping until the total number of deferred pages meets a threshold. Then, it unmaps these pages and invalidates their TLBs in batch. Different from the existing approaches, PMAlloc focuses on optimizing cross-NUMA memory deallocation within the user-space memory allocator. Our two-phase deallocation approach eliminates remote access induced by remote metadata management while ensuring crash consistency.

11 CONCLUSION

In the article, we design a novel allocator, named PMAlloc, to allocate/deallocate memory objects in persistent memory. PMAlloc 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. PMAlloc also provides NUMA-aware allocation and deallocation for multi-socket servers. Our experimental results demonstrate that PMAlloc can significantly improve allocator performance and space utilization. As persistent memory becomes more and more popular, we hope the various optimization techniques in PMAlloc will inspire the future generation of persistent memory systems.

ACKNOWLEDGMENTS

We sincerely thank the anonymous reviewers for their constructive suggestions.

REFERENCES

[1] Wilhelm Ackermann. 1928. Zum hilbertschen aufbau der reellen zahlen. *Math. Ann.* 99, 1 (1928), 118–133.

[2] Martin Aigner, Christoph M. Kirsch, Michael Lippautz, and Ana Sokolova. 2015. Fast, multicore-scalable, low-fragmentation memory allocation through large virtual memory and global data structures. *ACM SIGPLAN Not.* 50, 10 (2015), 451–469.

[3] Chloe Alverti, Vasileios Karakostas, Nikhita Kunati, Georgios Goumas, and Michael Swift. 2022. DaxVM: Stressing the limits of memory as a file interface. In *Proceedings of the 55th IEEE/ACM International Symposium on Microarchitecture (MICRO'22)*. 369–387.

[4] 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 ACM SIGMOD International Conference on Management of Data (SIGMOD'15)*. Association for Computing Machinery, 707–722.

[5] Josh Barnes and Piet Hut. 1986. A hierarchical O (N log N) force-calculation algorithm. *Nature* 324, 6096 (1986), 446–449.

[6] Emery D. Berger, Kathryn S. McKinley, Robert D. Blumofe, and Paul R. Wilson. 2000. Hoard: A scalable memory allocator for multithreaded applications. *ACM SIGPLAN Not.* 35, 11 (2000), 117–128.

[7] Bo Bernhardsson. 1991. Explicit solutions to the N-queens problem for all N. *ACM SiGART Bull.* 2, 2 (1991), 7.

[8] Kumud Bhandari, Dhruva R. Chakrabarti, and Hans-J. Boehm. 2016. Makalu: Fast recoverable allocation of non-volatile memory. *ACM SIGPLAN Not.* 51, 10 (2016), 677–694.

[9] Hans-Juergen Boehm. 1993. Space efficient conservative garbage collection. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'93)*. New York, NY, 197–206.

[10] 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 ACM SIGPLAN International Symposium on Memory Management (ISMM'20)*. 60–73.

[11] Zhichao Cao, Siying Dong, Sagar Vemuri, and David H. C. Du. 2020. Characterizing, modeling, and benchmarking RocksDB key-value workloads at Facebook. In *Proceedings of the 18th USENIX Conference on File and Storage Technologies (FAST'20)*. 209–223.

[12] 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 on Microarchitecture (MICRO'17)*. 191–203.

[13] 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 25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'20)*. 1077–1091.

[14] Zhangyu Chen, Yu Huang, Bo Ding, and Pengfei Zuo. 2020. Lock-free concurrent level hashing for persistent memory. In *Proceedings of theUSENIX Annual Technical Conference (ATC'20)*. 799–812.

[15] 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 Comput. Archit. News* 39, 1 (2011), 105–118.

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

[17] Intel Corporation. 2020. Persistent Memory Development Kit. Retrieved from http://pmem.io/

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

[19] 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'18)*. 271–282.

[20] 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)*. New York, NY, 115–127.

[21] Mohammad Dashti, Alexandra Fedorova, Justin Funston, Fabien Gaud, Renaud Lachaize, Baptiste Lepers, Vivien Quema, and Mark Roth. 2013. Traffic management: A holistic approach to memory placement on NUMA systems. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems*. New York, NY, 381–394.

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

[23] 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'20)*. 207–220.

[24] Matthias Diener, Eduardo H. M. Cruz, Marco A. Z. Alves, Philippe O. A. Navaux, Anselm Busse, and Hans-Ulrich Heiss. 2015. Kernel-based thread and data mapping for improved memory affinity. *IEEE Trans. Parallel Distrib. Syst.* 27, 9 (2015), 2653–2666.

[25] 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'19)*. 1–3.

[26] Jason Evans. 2021. Jemalloc. Retrieved from https://github.com/jemalloc/jemalloc/

[27] Fabien Gaud, Baptiste Lepers, Jeremie Decouchant, Justin Funston, Alexandra Fedorova, and Vivien Quéma. 2014. Large pages may be harmful on NUMA systems. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC 14)*. 231–242.

[28] Google Inc. 2021. tcmalloc. Retrieved from https://github.com/google/tcmalloc

[29] 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 *Proceedings of the USENIX Annual Technical Conference (ATC'19)*. USENIX Association, 913–928.

[30] Tom's Hardware. 2022. Samsung's Memory-Semantic CXL SSD Brings a 20x Performance Uplift. Retrieved from https://www.tomshardware.com/news/samsung-memory-semantic-cxl-ssd-brings-20x-performance-uplift

[31] Qingda Hu, Jinglei Ren, Anirudh Badam, Jiwu Shu, and Thomas Moscibroda. 2017. Log-structured non-volatile main memory. In *Proceedings of the USENIX Annual Technical Conference (ATC'17)*. 703–717.

[32] Xiameng Hu, Xiaolin Wang, Yechen Li, Lan Zhou, Yingwei Luo, Chen Ding, Song Jiang, and Zhenlin Wang. 2015. LAMA: Optimized locality-aware memory allocation for key-value cache. In *Proceedings of the USENIX Annual Technical Conference (ATC'15)*. USENIX Association, 57–69.

[33] Intel. 2018. 5-Level Paging and 5-Level EPT White Paper. Retrieved from https://www.intel.com/content/www/us/en/content-details/671442/5-level-paging-and-5-level-ept-white-paper.html

[34] Intel Inc. 2022. IPMCTL: A Command Line Interface (CLI) application for configuring and managing PMems. Retrieved from https://github.com/intel/ipmctl/
[35] Intel Inc. 2022. Processor Counter Monitor (PCM). Retrieved from https://github.com/intel/pcm/
[36] Intel Inc. 2023. *Intel® 64 and IA-32 Architectures Optimization Reference Manual.*
[37] Abdullah Al Raqibul Islam and Dong Dai. 2023. DGAP: Efficient dynamic graph analysis on persistent memory. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'23)*. Association for Computing Machinery, New York, NY.
[38] Keita Iwabuchi, Lance Lebanoff, Maya Gokhale, and Roger Pearce. 2019. Metall: A persistent memory allocator enabling graph processing. In *Proceedings of the IEEE/ACM 9th Workshop on Irregular Applications: Architectures and Algorithms (IA3'19)*. 39–44.
[39] Jemalloc. 2023. Jemalloc(3) Manual Page. Retrieved from https://jemalloc.net/jemalloc.3.html
[40] Hai Jin, Zhiwei Li, Haikun Liu, Xiaofei Liao, and Yu Zhang. 2020. Hotspot-aware hybrid memory management for in-memory key-value stores. *IEEE Trans. Parallel Distrib. Syst.* 31, 4 (2020), 779–792.
[41] Mark S. Johnstone and Paul R. Wilson. 1998. The memory fragmentation problem: Solved? *ACM SIGPLAN Not.* 34, 3 (1998), 26–36.
[42] Myoungsoo Jung. 2022. Hello bytes, bye blocks: PCIe storage meets compute express link for memory expansion (CXL-SSD). In *Proceedings of the 14th ACM Workshop on Hot Topics in Storage and File Systems (HotStorage'22)*. 45–51.
[43] Patryk Kaminski. 2009. NUMA aware heap memory manager. *AMD Devel. Centr.* (2009), 46.
[44] Sanidhya Kashyap, Changwoo Min, Kangnyeon Kim, and Taesoo Kim. 2018. A scalable ordering primitive for multicore machines. In *Proceedings of the 13th EuroSys Conference (EuroSys'18)*. Association for Computing Machinery, New York, NY.
[45] Wonbae Kim, Chanyeol Park, Dongui Kim, Hyeongjun Park, Young ri Choi, Alan Sussman, and Beomseok Nam. 2022. ListDB: Union of write-ahead logs and persistent skiplists for incremental checkpointing on persistent memory. In *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI'22)*. 161–177.
[46] Wook-Hee Kim, R. Madhava Krishnan, Xinwei Fu, Sanidhya Kashyap, and Changwoo Min. 2021. PACTree: A high performance persistent range index using PAC guidelines. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP'21)*. Association for Computing Machinery, 424–439.
[47] Joseph B. Kruskal. 1956. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proc. Amer. Math. Societ.* 7, 1 (1956), 48–50.
[48] Mohan Kumar Kumar, Steffen Maass, Sanidhya Kashyap, Ján Veselý, Zi Yan, Taesoo Kim, Abhishek Bhattacharjee, and Tushar Krishna. 2018. LATR: Lazy translation coherence. In *Proceedings of the 23rd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'18)*. New York, NY, 651–664.
[49] 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'98)*. 176–185.
[50] 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'17)*. 257–270.
[51] 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'19)*. 462–477.
[52] Daan Leijen. 2019. MiMalloc Benchmarks. Retrieved from https://github.com/daanx/mimalloc-bench
[53] Daan Leijen, Benjamin Zorn, and Leonardo de Moura. 2019. MiMalloc: Free list sharding in action. In *Proceedings of the 17th Asian Symposium on Programming Languages and Systems (APLAS')19*. Springer, 244–265.
[54] Lenovo. 2018. Memcached-PMEM. Retrieved from https://github.com/lenovo/memcached-pmem/
[55] Baptiste Lepers, Vivien Quéma, and Alexandra Fedorova. 2015. Thread and memory placement on NUMA systems: Asymmetry matters. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC'15)*. 277–289.
[56] Zhenxin Li, Bing Jiao, Shuibing He, and Weikuan Yu. 2022. PhaST: Hierarchical concurrent log-free skip list for persistent memory. *IEEE Trans. Parallel Distrib. Syst.* 33, 12 (2022), 3929–3941.
[57] Jihang Liu, Shimin Chen, and Lujun Wang. 2020. LB+ Trees: Optimizing persistent index performance on 3DXPoint memory. *Proc. VLDB Endow.* 13, 7 (2020), 1078–1090.
[58] Baotong Lu, Xiangpeng Hao, Tianzheng Wang, and Eric Lo. 2020. Dash: Scalable hashing on persistent memory. *Proc. VLDB Endow.* 13, 8 (2020), 1147–1161.
[59] 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'21)*. 1–16.
[60] MicroQuill Inc. 2014. shbench. Retrieved from http://www.microquill.com/

[61] Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, and Kunle Olukotun. 2008. STAMP: Stanford transactional applications for multi-processing. In *Proceedings of the IEEE International Symposium on Workload Characterization*. IEEE, 35–46.

[62] 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 1st ACM SIGOPS Conference on Timely Results in Operating Systems (TRIOS'13)*. 1–17.

[63] Ismail Oukid, Daniel Booss, Adrien Lespinasse, Wolfgang Lehner, Thomas Willhalm, and Grégoire Gomes. 2017. Memory management techniques for large-scale persistent-main-memory systems. *Proc. VLDB Endow.* 10, 11 (2017), 1166–1177.

[64] 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 International Conference on Management of Data (SIGMOD'16)*. 371–386.

[65] Xing Pan, Yasaswini Jyothi Gownivaripalli, and Frank Mueller. 2016. TintMalloc: Reducing memory access divergence via controller-aware coloring. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS'16)*. 363–372.

[66] Mihail Popov, Alexandra Jimborean, and David Black-Schaffer. 2019. Efficient thread/page/parallelism autotuning for NUMA systems. In *Proceedings of the ACM International Conference on Supercomputing*. 342–353.

[67] 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.

[68] Andy Rudoff. 2020. Persistent memory programming without all that cache flushing. Retrieved from https://www.snia.org/educational-library/persistent-memory-programming-without-all-cache-flushing-2020

[69] 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'14)*. 1–16.

[70] 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'06)*. 84–94.

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

[72] Haris Volos, Andres Jaan Tack, and Michael M. Swift. 2011. Mnemosyne: Lightweight persistent memory. *ACM SIGARCH Comput. Archit. News* 39, 1 (2011), 91–104.

[73] Mehul Wagle, Daniel Booss, Ivan Schreter, and Daniel Egenolf. 2015. NUMA-aware memory management with in-memory databases. In *Proceedings of the Technology Conference on Performance Evaluation and Benchmarking (TPCTC'15)*. Springer, 45–60.

[74] Qing Wang, Youyou Lu, Junru Li, and Jiwu Shu. 2021. Nap: A black-box approach to NUMA-aware persistent memory indexes. In *Proceedings of the 15th USENIX Symposium on Operating Systems Design and Implementation (OSDI'21)*. USENIX Association, 93–111.

[75] Rui Wang, Shuibing He, Weixu Zong, Yongkun Li, and Yinlong Xu. 2022. XPGraph: XPline-friendly persistent memory graph stores for large-scale evolving graphs. In *Proceedings of the 55th IEEE/ACM International Symposium on Microarchitecture (MICRO'22)*. 1308–1325.

[76] 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'95)*. Springer, 1–116.

[77] 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'21)*. 141–153.

[78] Lingfeng Xiang, Xingsheng Zhao, Jia Rao, Song Jiang, and Hong Jiang. 2022. Characterizing the performance of Intel Optane persistent memory: A close look at its on-DIMM buffering. In *Proceedings of the 17th European Conference on Computer Systems (EuroSys'22)*. 488–505.

[79] 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'20)*. 169–182.

[80] Diyu Zhou, Yuchen Qian, Vishal Gupta, Zhifei Yang, Changwoo Min, and Sanidhya Kashyap. 2022. ODINFS: Scaling PM performance with opportunistic delegation. In *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI'22)*. USENIX Association, 179–193.

Received 26 November 2022; revised 8 September 2023; accepted 21 January 2024