2023 IEEE 41st International Conference on Computer Design (ICCD) | 979-8-3503-4291-8/23/$31.00 ©2023 IEEE | DOI: 10.1109/ICCD58817.2023.00036

PMA: A Persistent Memory Allocator with High Efficiency and Crash Consistency Guarantee

Xiangyu Xiang, Yu Hua*, Hao Xu
WNLO, Huazhong University of Science and Technology, Wuhan, Hubei, China
*Corresponding author: Yu Hua
E-mail: {xyxiang, csyhua, hao_xu}@hust.edu.cn

***Abstract*—Byte-addressable persistent memory (PM) exhibits salient features of low latency and high capacity. PM can be memory-mapped to the virtual address space of a process and be directly accessed via load and store instructions. Persistent memory allocator is a fundamental building block in PM-oriented programs, which provides dynamic memory allocation/deallocation primitives for developers to efficiently and safely leverage the PM. Different from DRAM allocators, a PM allocator needs to guarantee the integrity and consistency of metadata in the face of a crash. To this end, we propose a high-efficiency PM allocator, called PMA, with crash consistency guarantee. PMA uses a two-level memory management strategy and sets up a private memory pool for each thread to achieve low fragmentation and high concurrency. Furthermore, PMA employs per-thread write-ahead undo log to protect the integrity and consistency of metadata against crashes. PMA also designs a lightweight persistent pointer to reference an allocated persistent memory object across runtimes. PMA is implemented as an easy-to-use library that is independent of specific PM platforms. Extensive evaluation results on a real PM platform demonstrate the efficiency and efficacy of our proposed PMA, compared with state-of-the-art log-based PM allocators.**

***Index Terms*—Persistent Memory, Memory Management, Memory Allocator, Crash Consistency**

I. Introduction

Persistent memory (PM) coalesces the high performance of DRAM and the persistence of hard disks, blurring the boundary between main memory and external storage [3], [4]. It provides near-DRAM latency and much larger capacity while also retaining data after power-off. Moreover, a user process can directly access PM via load/store instructions in the user space, without kernel involvement [1]. PM creates new opportunities for constructing a new storage architecture and building high-efficiency in-memory applications. Although the first commercial PM product, Intel Optane DC PMM [5], has been recently discontinued, researches on PM are still going on. Our work does not rely on Optane DC PMM and can be used on any other non-volatile memory platform that supports memory-mapping.

Operating system manages PM via PM file systems [2], [6], [7], which are similar to traditional hard disk file systems. The main difference is that a user process can memory-map a PM file to its virtual address space without kernel page cache through the direct access (DAX) mechanism. The process can further access PM in the user space, eliminating the high overheads of crossing the user-kernel boundary and going through the storage stack [7]. But the challenge is that the user process needs to efficiently manage the PM space on its own to meet the dynamic memory requirements from multiple threads. A PM allocator takes the responsibility for dynamically allocating and reclaiming persistent memory according to the program's requirements. It provides easy-to-use interface for application developers to efficiently and safely exploit the PM.

Traditional DRAM allocators, such as *Hoard* [9], *jemalloc* [11] and *TCMalloc* [10], have achieved low latency, high scalability, and low fragmentation. They can be applied to PM due to its byte-addressable property the same as DRAM. However, if a system crash occurs, the metadata of the allocator could be corrupted and would remain at an inconsistent state even after the system restarts due to the persistence of PM [8]. The corruption of metadata leads to serious problems, like memory leak or deadlocks that cannot be solved by restarting neither the program nor the system. Therefore, a PM allocator needs to guarantee the integrity and consistency of metadata in the face of a crash. Moreover, a persistent pointer is necessary to persistently reference an allocated memory block across multiple runs of a program, which will be discussed in details in Section II-C.

In this paper, we propose a high-performance and crash-consistent persistent memory allocator, called PMA. PMA uses a two-level memory management strategy: It divides the whole PM space into multiple fixed-size superblocks (SBs) and divides each superblock into several smaller memory blocks. To mitigate the contentions among multiple user threads, PMA sets up a private memory pool for each thread. In most cases, a user thread can easily obtain the free memory blocks from its own memory pool, which is inaccessible by other threads. Thus, no lock is needed. PMA employs write-ahead undo log to protect the integrity and consistency of metadata against crashes. It reserves a log region for each thread separately to make them work concurrently. The log space overhead is acceptable since the amount of metadata is very limited. PMA implements a lightweight offset-based persistent pointer to persistently reference an allocated persistent memory object even when the process is terminated or the PM file is remapped to a different address. PMA guarantees that the user process can always access a persistent memory object correctly by

This work was supported in part by National Natural Science Foundation of China (NSFC) under Grant No. 62125202, U22B2022 and 61821003.

2576-6996/23/$31.00 ©2023 IEEE
DOI 10.1109/ICCD58817.2023.00036

dereferencing its persistent pointer.

We evaluate PMA on a real PM platform equipped with Optane DC PMM and compare PMA with state-of-the-art log-based PM allocators, including *libpmemobj* [1], *nvm_malloc* [15] and *NVAlloc-log* [19], by using multiple metrics. Extensive experimental results demonstrate that PMA can achieve up to 3.4x, 1.6x and 19.1x higher throughput than *libpmemobj*, *nvm_malloc* and *NVAlloc-log*, respectively. PMA also reduces the tail latency of small allocation by orders of magnitude. Furthermore, PMA recovers 512 uncommitted transactions in approximately 120 microseconds.

In summary, this paper makes the following contributions:

- We propose a persistent memory allocator PMA with high throughput, low latency and crash consistency.
- We implement a lightweight offset-based persistent pointer that can reference a persistent memory object across runtimes.
- We evaluate PMA on a real PM platform and compare it with state-of-the-art log-based PM allocators via multiple performance metrics.

II. Background and Motivation

A. PM and DAX

Byte-addressable persistent memory (PM) has significantly higher capacity and lower energy consumption than DRAM, whereas its access latency is on the same order of magnitude like DRAM [3], [5]. PM is connected on the memory bus and can be directly accessed by the user process without the kernel involvement [1]. With the aid of PM, programs can directly persist data in memory, eliminating data serialization/deserialization and going through the storage stack, which would significantly reduce the software overheads.

PM is usually mounted to the file system and managed in the form of files [2], [6], [7]. As shown in Figure 1, there are typically two approaches for applications to access PM. First, traditional applications designed for hard disks can access PM through POSIX I/O operations (e.g., read()/write()) without any modification, which simply treats the PM as a faster disk. Second, with the help of the direct access (DAX) mechanism [1], applications can memory-map a PM file to the virtual address space without kernel page cache and directly access the PM via load/store instructions. DAX mechanism eliminates the overhead of trapping into OS kernel, thus significantly improving the access performance. According to our experimental results on a real PM platform, using the DAX-enabled memory-mapping approach to write/read 1GB data to/from the PM can reduce the execution time by 81% and 56%, respectively (as shown in Table I).

TABLE I. The execution time of writing/reading 1GB data to/from the PM through POSIX I/O operations or DAX-enabled memory-mapping.

	I/O operations	DAX-enabled memory-mapping
Writing Time (s)	19.54	3.81
Reading Time (s)	1.42	0.63

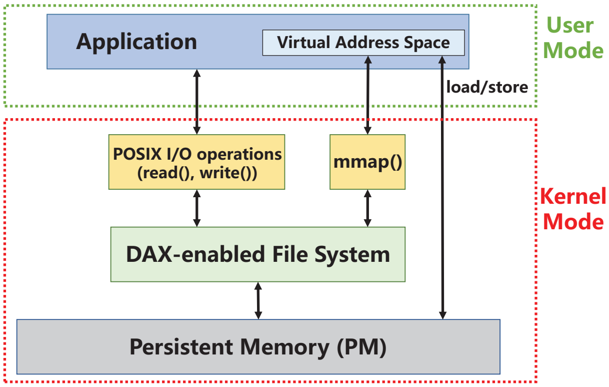


Fig. 1. The two access paths to PM. One way is through the I/O operations, and the other is DAX-enabled **mmap()**.

When using the DAX-enabled memory-mapping, the PM file is guaranteed to be mapped to contiguous virtual address space. This space is usually called a *persistent heap* and the mapped PM file is called a *heap file*. The PM allocator takes the responsibility for managing the *persistent heap*. It dynamically allocates free memory block from the *persistent heap* when a user thread needs and reclaims the memory block when it is freed.

B. Crash Consistency

Despite using the DAX mechanism, applications cannot bypass the CPU cache when writing to PM. The data written to PM is not guaranteed to be persistent immediately due to possibly residing in the volatile CPU cache. The data will be persisted only when it is flushed back to PM from the CPU cache. However, if a crash occurs before the data is flushed back, the data's newest version will be lost. Some data may be partially persisted in arbitrary orders due to the random cache line evictions, which unfortunately breaks the integrity and consistency of data [8]. For example, if there is a string *"AAA$\cdots$A"* (128 letters) on PM and we change it to *"BBB$\cdots$B"* (128 letters) in a program. This modification will take effect after three steps: (1): The string is copied to the CPU cache for faster access and occupies two cache lines (since the size of a cache line is typically 64 bytes). (2): The string in the cache is changed to *"BBB$\cdots$B"*. (3): The two cache lines are flushed back to PM and thus the string *"BBB$\cdots$B"* is persisted. However, if a crash occurs during step 1 or step 2, the modification will be completely lost and the string on PM is still *"AAA$\cdots$A"*. If a crash occurs during step 3, perhaps only one of the cache lines is flushed back to PM while the other is lost, and the order is random. Finally the string on PM may become *"AA$\cdots$ABB$\cdots$B"* or *"BB$\cdots$BAA$\cdots$A"*, which are inefficient.

There are some instructions like CLWB and SFENCE to explicitly flush cache lines back in the specified order [1]. Specifically, CLWB means cache line writing back. It carries an address as the parameter and flushes the cache line that contains such address back to the PM immediately. SFENCE is used to guarantee the execution order of multiple CLWBs.

The CLWB after a SFENCE can be performed only after the CLWB before the SFENCE is completed. Therefore, programming with PM requires the developers to properly persist data with these instructions to ensure the integrity and consistency of data. To ease the developer's burden and hide the underlying details about data persistence, PMA provides some high-level and easy-to-use interfaces (as shown in Section III-A).

C. *Persistent Pointer*

Due to the byte-addressable property of PM, a user process can directly access it through pointers. When a PM allocator allocates a free memory block for a user thread, it will return the start address of this memory block and the user thread can further access this region through the pointer to it. However, traditional pointers reference the virtual addresses, which are valid only within the lifetime of the process. After the process terminates or restarts, the virtual addresses become invalid. Besides, the *heap file* may be mapped to different virtual addresses in different runs of a program due to the *address space layout randomization (ASLR)*. Therefore, we cannot permanently reference a memory region on the *persistent heap* via its virtual address. Hence a persistent pointer is necessary.

Figure 2 illustrates the aforementioned issue when a *heap file* is mapped to different virtual addresses in different runs of a program. The *heap file* is mapped to 0x7f31c7600000 at first, then a memory block at 0x7f31c7600800 is allocated and this address is assigned to the *ptr*. Hence, the user process can access the allocated memory block with *ptr*. However, when the program restarts, the *heap file* is remapped to 0x7f312ab00000 and the previously allocated memory block should be at 0x7f312ab00800. Hence the previous *ptr* becomes invalid. If we still access the *persistent heap* with *ptr*, some unpredictable errors would happen.

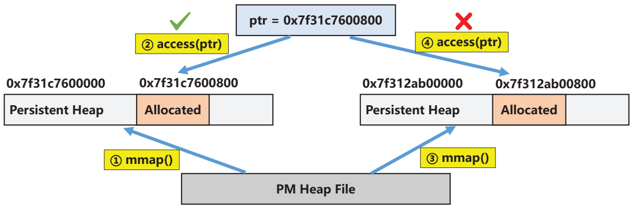


Fig. 2. The heap file is mapped to different virtual addresses in different runs of a program.

III. The Design of PMA

We design and implement PMA as an easy-to-use user-level library. It is built on the top of a kernel file system that supports DAX-enabled memory-mapping (e.g., ext4-DAX). PMA is independent of any specific PM platform and can be used for any non-volatile memory that is byte-addressable. A PMA *heap file* is completely self-contained and can be moved, copied, or transmitted among multiple machines, as long as the user process accesses the *heap file* through PMA.

A. *Overview and Programming Model*

Figure 3 depicts the high-level architecture of PMA. The PM *heap file* is memory-mapped to the contiguous virtual address space via the memory-mapping system calls provided by the DAX-enabled kernel file system (e.g., **mmap()** in ext4-DAX). PMA further manages the *persistent heap* in the user space and allocates free memory blocks from the *persistent heap* when required.

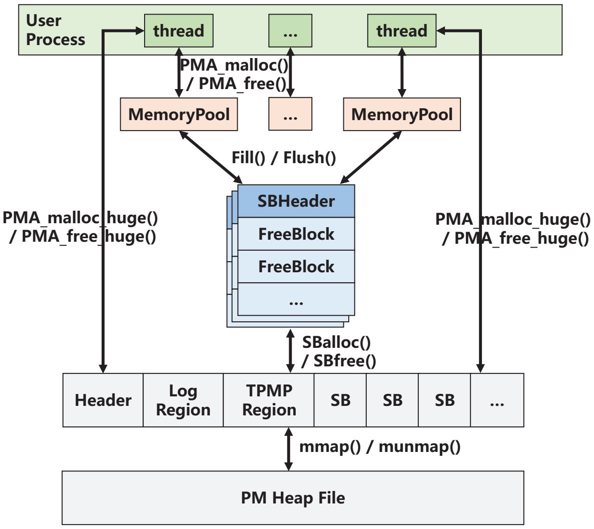


Fig. 3. The framework of PMA.

The *persistent heap* is divided into four main regions: the header region, the log region, the thread-private memory pool (TPMP) region, and the superblock region. Specifically, the header region stores some essential metadata for the whole *persistent heap*, such as a *magic number* marking it as a PMA *heap file*, the starting address and the length of the *persistent heap*, etc. The log region contains the write-ahead undo log for metadata recovery after a crash (Section III-F). The TPMP region consists of some TPMP descriptors (Section III-D). The superblock region is divided into multiple superblocks in the equal size (16KB by default). PMA takes the superblocks as basic units when managing the *persistent heap* and then manages the interior of each superblock separately at a finer granularity (Section III-C).

In order to mitigate the memory allocation contentions among multiple user threads, PMA sets up a TPMP for each thread (Section III-D). The thread obtains memory directly from its TPMP, which is inaccessible by other threads and thus no concurrency control is necessary. The TPMP will be filled up with the globally accessible superblocks if exhausted. PMA establishes a TPMP for a user thread automatically when the thread first calls for memory. All the TPMPs will be flushed back to the global superblock region before PMA exits. It is worth noting that the TPMPs only hold small memory blocks (no more than half the size of a superblock), and larger memory blocks are allocated directly from the global superblock region, thus bypassing the TPMP, as discussed in Section III-E.

PMA provides 8 interface functions for developers to obtain memory from the *persistent heap* flexibly and safely, which are listed in Table II. When using PMA, the user process

TABLE II. The APIs provided by PMA.

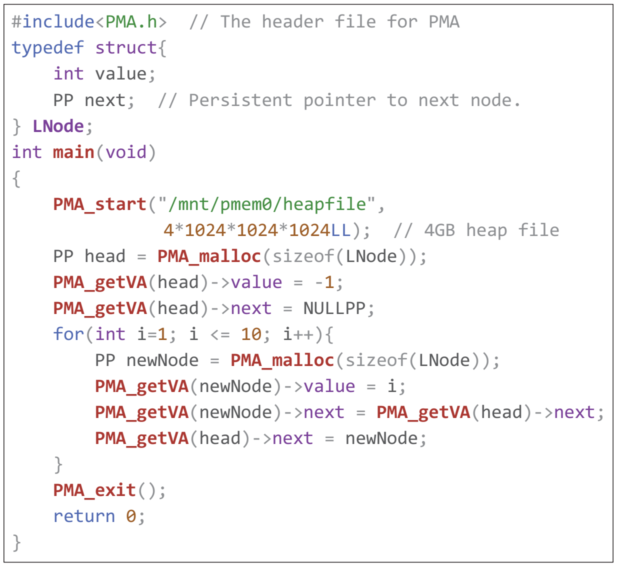
Function	Description
PMA_start (heapfile, size)	Start the allocator and memory-map the *heap file* with the given *size*.
PMA_exit ()	Unmap the *heap file* and exit the allocator safely.
PMA_malloc (size)	Allocate a free memory block no smaller than the given *size* and return its persistent pointer.
PMA_free (pp)	Reclaim the memory block referenced by the given persistent pointer *pp*.
PMA_getPP (va)	Transform the given virtual address *va* to a persistent pointer *pp*.
PMA_getVA (pp)	Transform the given persistent pointer *pp* to a virtual address *va*.
PMA_persist (addr, len)	Persist the memory region from *addr* to *addr+len*.
PMA_barrier ()	Restrict the execution order of multiple PMA_persist().

first invokes PMA_start() to memory-map the *heap file*. If the *heap file* does not exist or is empty, PMA creates a new *heap file* of the given size and initializes the basic structure of the *persistent heap*. After start up, the user process can invoke PMA_malloc() to allocate a free memory block from the *persistent heap* or PMA_free() to reclaim it. PMA_malloc() returns the persistent pointer of the allocated memory block. The user process can transform the persistent pointer to a virtual address pointer via PMA_getVA() or transform the virtual address pointer to a persistent pointer via PMA_getPP(). PMA_persist() is used to persist data in the given range immediately and PMA_barrier() restricts the execution order of multiple PMA_persist(). If a PMA_barrier() is added between two PMA_persist(), it is guaranteed that the latter PMA_persist() can be performed only after the previous one finishes. Otherwise, the executions of multiple PMA_persist() may be disordered.

We further present a program as an example in Figure 4 to show the programming model of PMA. In this example, we construct a linked list on the *persistent heap* with PMA. It is similar to construct a linked list on the DRAM by using malloc()/free() in the C standard library. The main difference is that the linked list on the *persistent heap* is persistent and can still be accessed via persistent pointers even after the system reboots.

B. Size Class

In order to normalize the memory allocation sizes and limit the memory fragmentation rate to an acceptable range, we predefined 41 standard *size classes*, such as 8B, 10B, 12B, 14B, 16B, 20B, 24B, 28B, 32B, etc. Other sizes will be aligned up to the nearest standard size. For example, if the user thread requires a 30-byte block, PMA will allocate a 32-byte free block, even though there is a 2-byte internal fragmentation. We argue this is acceptable due to the large capacity of PM. Moreover, we eliminate the external fragmentations and limit the internal fragmentation rate to no more than 20% by appropriately generating standard *size classes* [12].



```
#include<PMA.h>  // The header file for PMA
typedef struct{
    int value;
    PP next;  // Persistent pointer to next node.
} LNode;
int main(void)
{
    PMA_start("/mnt/pmem0/heapfile",
              4*1024*1024*1024LL);  // 4GB heap file
    PP head = PMA_malloc(sizeof(LNode));
    PMA_getVA(head)->value = -1;
    PMA_getVA(head)->next = NULLPP;
    for(int i=1; i <= 10; i++){
        PP newNode = PMA_malloc(sizeof(LNode));
        PMA_getVA(newNode)->value = i;
        PMA_getVA(newNode)->next = PMA_getVA(head)->next;
        PMA_getVA(head)->next = newNode;
    }
    PMA_exit();
    return 0;
}
```

Fig. 4. An instance of constructing a persistent linked list on the persistent heap with PMA.

C. Superblock

Excluding metadata, the *persistent heap* is divided into multiple superblocks of the same size (16KB in our implementation), which are the basic units of global memory management. Each superblock contains a descriptor in the head (as shown in Figure 5), which stores the metadata of this superblock, including the *size class* of this superblock, the number of free blocks in this superblock, the index of the first free block, and the pointer to the prior/next superblock. Apart from the descriptor, the remaining space of the superblock is divided into several fixed-size small blocks depending on the superblock's *size class*. The *size class* of the superblock determines the size of the internal small blocks. For instance, the superblock belonging to *size class 0* will be divided into several 8-byte blocks and the superblock belonging to *size class 8* will be divided into several 32-byte blocks. Therefore, all the blocks in the same superblock have the same size according to the superblock's *size class*, while the blocks in two different superblocks may have different sizes.

Free blocks in a superblock are coalesced together into a linked list. Both the header pointer and length of the list are stored in the superblock's descriptor. Even though the bitmap has been widely used to mark free or occupied memory blocks, we found that the linked list is more efficient. Because we can obtain a free block from the head of the list with $O(1)$ time complexity, whereas the bitmap technique needs to find a free block by scanning the bitmap with $O(n)$ time complexity (n denotes the size of the bitmap).

According to the number of free blocks inside, a superblock can be in one of the following three states:

- **Empty**. An empty superblock is entirely free, which has not been initialized and does not belong to any *size class* (the *SizeClass* field in its descriptor is -1).
- **Partial**. A partial superblock has been initialized and belongs to a certain *size class*. It is divided into multiple small blocks and some have been allocated, but not all.
- **Full**. All blocks in a full superblock have been allocated and there are no available free blocks in this superblock.

We use doubly linked lists to manage all superblocks. Superblocks in the same state and the same *size class* are added to the same list. Specifically, all the empty superblocks are linked into one list since they do not belong to any *size class*, whereas partial/full superblocks are linked into several separate lists based on their *size classes*. Partial/full superblocks belonging to the same *size class* are added to the same list. Furthermore, the state of a superblock would change as PMA runs, and hence it would be moved from one list to another. For instance, when an empty superblock is allocated and initialized with *SizeClass=2*, it will be removed from the *Empty List* and be added to the *Partial[2] List*. When all the blocks are allocated at some point, it will be moved from the *Partial[2] List* to the *Full[2] List*. Note that this process is reversible when freeing blocks.

```
typedef struct{ //The descriptor of a superblock
    int SizeClass; //The size class it belongs to (from 0 to 40, or -1 for empty SB)
    int BlockSize;      //The size of small blocks inside
    int FirstFreeBlock; //The index of the first free block
    int NumOfFreeBlocks;//The number of free blocks
    PP  prior;          //Pointer to the prior SB in the SB list
    PP  next;           //Pointer to the next SB in the SB list
} SBdescriptor;
```

Fig. 5. The data structure of a superblock descriptor.

D. Thread-Private Memory Pool

To achieve high concurrency, PMA sets up a thread-private memory pool (TPMP) for each user thread. A TPMP consists of some free blocks that are only visible and accessible to a certain thread. These free blocks are coalesced together into several linked lists based on their *size classes* and the header pointers are owned by this thread privately, so other threads cannot access these free blocks. The descriptor of a TPMP is defined as shown in Figure 6. There is a separate free block list for each standard *size class* (41 lists for 41 *size classes* in total). Both the header pointer and the length of the free block list are stored in the TPMP descriptor.

The size of one TPMP descriptor is 664B. PMA reserves 640KB space for the TPMP region and thus up to 986 user threads can be supported. When a thread calls for memory for the first time, PMA occupies a free TPMP for this thread and allocates some free blocks from the global superblocks to fill the TPMP. Future memory allocation requests can be handled by the TPMP without global memory allocation, and thus no concurrency control is required.

TPMP significantly reduces the overhead of concurrency control by reducing the occurrence of global memory allocation. For example, if we fill a TPMP with 1,024 free blocks every time it runs out, the following 1,024 allocation requests can be easily handled without concurrency control. In such case, the occurrence rate of global memory allocation is less than 0.1%. Hence the overhead of concurrency control is significantly reduced.

```
typedef struct{ //The descriptor of a thread-private memory pool
    pthread_t tid;  //The owner thread of this TPMP.
    struct{
        PP header;  //Header pointer to the free block list.
        size_t length; //Length of the free block list.
    } freeLists[41];//One free list for one size class.
} TPMP;
```

Fig. 6. The data structure of a TPMP descriptor.

E. Allocation and Deallocation

Allocation. When a user thread invokes PMA_malloc() to allocate a free block with a specified size, PMA first checks whether the allocation size is too large. If the allocation size is larger than half the size of a superblock (the maximum size in our standard *size classes*), this allocation will bypass the TPMP and be achieved by allocating several contiguous superblocks. For example, a 60KB allocation request will receive 4 contiguous superblocks (64KB in total). Small allocation request will be aligned up to the nearest standard *size class* and allocated from the TPMP. If the TPMP is empty, PMA will allocate some partial superblocks in the same *size class* to fill it. If there is no suitable partial superblock, PMA will allocate an empty superblock and add it to such *size class*.

Deallocation. When a user thread invokes PMA_free() to reclaim a block, PMA needs to find the superblock that contains such block at first. This can be achieved elegantly by aligning the address down to 16KB (the size of a superblock), since the start address of a superblock is guaranteed to be a multiple of 16KB. PMA further obtains the *size class* of this block in the superblock descriptor and adds the block to the TPMP. If the TPMP is full (the number of free blocks inside exceeds a threshold), PMA will flush half of the free blocks back to the global superblocks. Like allocation, the deallocation requests of huge blocks will also be handled bypassing the TPMP. Moreover, there is an issue that needs to be clarified: If a memory block allocated by thread T1 is deallocated by thread T2, it will be added to T2's TPMP rather than T1's, since T2 cannot access T1's TPMP. This would not cause any problem since the block is no longer needed by T1 and will be returned back to a global superblock eventually no matter if it is in T1's TPMP or T2's.

F. Per-Thread Log

When allocating memory, PMA employs write-ahead undo log to guarantee the integrity and consistency of metadata, which writes the old version of metadata to the log region before updating the metadata. Hence the *persistent heap* can be rolled back to the nearest consistent state based on the undo log after a system crash. Moreover, PMA reserves an individual log region for each thread to allow them to work concurrently. Figure 7 shows the layout of the entire log region. The whole log region (256KB) is divided into 1,024 subregions (256B) and thus at most 1,024 threads can run concurrently. Each subregion contains the *TID* (the owner thread of this subregion), the state of the transaction (RUNNING or COMMITTED), the type of the transaction (which will be discussed in the following paragraph), and the buffer to hold the old version of metadata.

There are three kinds of transactions in PMA. We will analyze the metadata that need to be updated in each kind of transaction below:

1) Moving a superblock from one list to another (as previously mentioned in Section III-C). In this transaction, five superblock descriptors need to be updated atomically: the superblock itself, the prior and next superblock in the old list, and the prior and next superblock in the new list.
2) Adding a free block to a TPMP from a global superblock or vice versa. In this transaction, both the superblock descriptor and the TPMP descriptor need to be atomically updated.
3) Allocating/Deallocating a block from/to a TPMP. In this transaction, only the TPMP descriptor needs to be atomically updated.

According to the above analysis, there are at most five metadata items to be updated in a transaction. PMA divides the buffer holding the old version of metadata into five entries and each entry contains the address, length, and old version value of the metadata (as shown in Figure 7). PMA reserves a 32B field in each entry to hold the old version value of metadata, because the size of a superblock descriptor is 32B. Although the size of a TPMP descriptor is larger than 32B (actually 664B), only one of the 41 free block lists in the TPMP will be modified each time and thus only 16B metadata will be updated.

The execution process of the transaction in PMA is summarized below:

1) Write the transaction type and old version metadata to the log region and persist the log region.
2) Set the state of the transaction to RUNNING.
3) Update the metadata and persist the new version.
4) Set the state of the transaction to COMMITTED.

Note that a PMA_barrier() is required between any two steps to make sure that the undo log has been persisted before modifying the metadata. If a system failure occurs, PMA will scan the log region after restarting to find out uncommitted transactions remaining at RUNNING state and roll them back according to the write-ahead undo log.

G. Persistent Pointer

We design a lightweight offset-based persistent pointer based on the insight that even though the virtual address of a certain position would change after memory remap, its offset relative to the start of the *persistent heap* keeps constant. Hence, a persistent pointer in PMA holds the offset of a memory block relative to the start of the *persistent heap*. A persistent pointer can be transformed to a virtual address in real time by adding the current base address of the *persistent heap*, and vice versa. Existing PM allocators simply return the virtual address of the allocated memory block. The developers need to manually transform the virtual address into offset by subtracting the start address of the persistent heap. PMA hides these details by providing the persistent pointer abstraction.

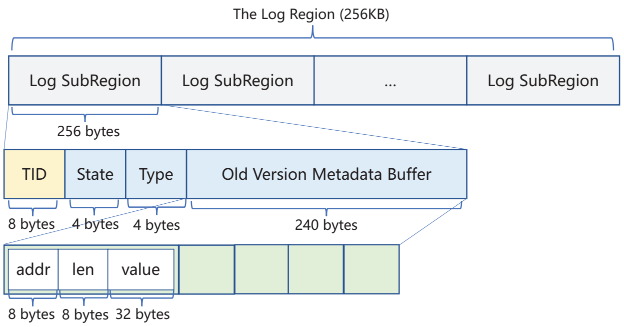


Fig. 7. The layout of the log region.

IV. Performance Evaluation

A. Experimental Setup

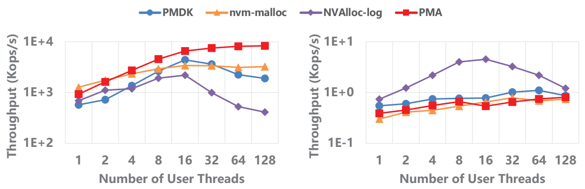
Our experiments run on a Linux server (Ubuntu 18.04) equipped with 768GB Intel Optane DC PMM 100 Series (128GB × 6) and 192GB DRAM (16GB × 12). There are two CPU sockets in the server and each socket is equipped with a 26-core Intel Xeon Gold 6230R at 2.10GHz. Each core has 32KB L1 data/instruction cache and 1MB L2 cache. The cores of one processor share 35.75MB L3 cache. The Optane DC PMMs are connected to the same CPU socket and configured in interleaved and App Direct Modes [3].

PMA is compared with the state-of-the-art log-based PM allocators, including *libpmemobj*, *nvm_malloc* and *NValloc-log*. The *libpmemobj* is a transactional object store library in the *persistent memory development kit (PMDK)* [1] developed by Intel. The *nvm_malloc* [15] is an allocator designed for non-volatile memory (NVM) and guarantees the ACID (atomicity, consistency, isolation, durability) of metadata. There are two versions of *NVAlloc* [19]: *NVAlloc-GC* and *NVAlloc-log*. The *NVAlloc-GC* is based on offline garbage collection, while the *NVAlloc-log* is based on write-ahead log. We use the log version for fair comparison. The comparisons with other GC-based NVM allocators like *Makalu* [17] and *Ralloc* [18] are out of the scope in this paper.

B. Allocation/Deallocation Performance

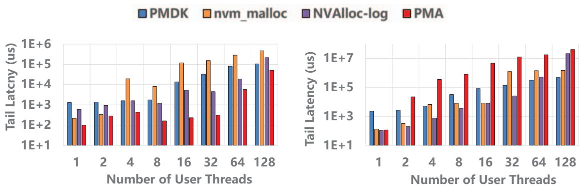
We spawn multiple threads in the test program to concurrently allocate 10^6 fixed-size memory blocks in total and then deallocate them all, repeating for 100 iterations. We record the execution time of each allocation operation and the total execution time of the program.

Throughput. We use the total number of allocation and deallocation operations divided by the total execution time of the program as the throughput. The evaluation results of the four allocators are shown in Figure 8. The X-coordinate is the



(a) Allocating 64B
(b) Allocating 32KB

Fig. 8. Throughput of allocation and deallocation operations.



(a) Allocating 64B
(b) Allocating 32KB

Fig. 9. Tail latency of allocation.

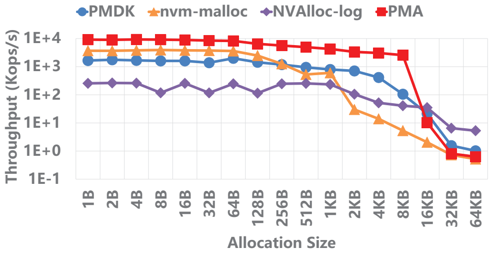


Fig. 10. Throughput of various allocation sizes.

number of user threads and the Y-coordinate is the throughput (kilo-operations per second, log10 scaled). Both the small allocation size (64B) and large allocation size (32KB) are evaluated. PMA achieves better performance than the competitors in the small allocation test, as shown in Figure 8(a). It delivers up to 3.4x, 1.6x and 19.1x higher throughput than *libpmemobj*, *nvm_malloc* and *NVAlloc-log*, respectively. The performance improvements come from our TPMP mechanism, which not only shortens the critical path of the allocation process, but also reduces the overheads of concurrency control. However, in the large allocation, as shown in Figure 8(b), PMA shows similar performance to *nvm_malloc*. Because 32KB is larger than the maximum standard *size class* (8KB) in PMA and thus the TPMP is disabled. This will be improved in our future work.

Tail Latency. We regard the maximum execution time of all allocation operations as the tail latency of allocation, as shown in Figure 9. PMA shows orders of magnitude lower tail latency than its competitors in the small allocation (Figure 9(a)), but higher tail latency in the large allocation (Figure 9(b)). This is because PMA bypasses the TPMP and uses mutex locks to achieve concurrency control in large allocation, which causes the blocking of threads and introduces high tail latency.

Allocation Size Sensitivity. We evaluate the size sensitivity of allocators in terms of various allocation sizes. As shown in Figure 10, PMA delivers higher and more stable throughput than the three competitors, when the allocation size is smaller than 8KB. This is because the sizes no larger than 8KB will be aligned to a standard *size class* and benefit from the TPMP mechanism. However, PMA suffers from a sharp drop of performance when the allocation size exceeds 8KB, which is exactly the largest standard *size class* in our implementation. This can be improved by expanding the standard *size classes* or including the large allocation into the TPMP, which will be explored in our future research. Nevertheless, the performance of *PMDK* and *nvm_malloc* decreases rapidly as well when the allocation size grows.

C. Log and Flush Overheads

To evaluate the overheads of log operations and flush instructions, we remove these functions from PMA and compare it with the original version. We employ multiple threads to concurrently allocate 10^6 memory blocks of 64B and deallocate them after, repeating for 100 iterations. Figure 11 shows the execution time of different PMA versions. The *PMA* represents a fully functional allocator, the *PMA-nolog* represents an allocator without write-ahead log, and the *PMA-noflush* represents an allocator without persistence (no log as well). The experimental results demonstrate that the log operations account for more than 60% of the execution time and the flush operations account for more than 95%. The write-ahead log introduces more than double writes to the PM, since some metadata are packed into the log entry as well (Section III-F). The flush operations are so costly because the writing speed of PM is still much slower than the CPU cache. But both the log and the flush operations are necessary for crash consistency. Metadata is at risk of being corrupted without the log or the flush operations.

D. Recovery Performance

Upon system crash, PMA can automatically roll back all uncommitted transactions and restore the metadata to the nearest consistent state based on the write-ahead undo log after restart. To evaluate the recovery performance of PMA, we remove all TX_COMMIT() in the program, which results in some transactions remaining uncommitted after the program exits. We further restart the program and the recovery routine will be invoked automatically in PMA_start(). It will scan the log region and figure out all uncommitted transactions based on the *state* flag in the log entry. Once an uncommitted transaction is found, PMA restores the metadata that has been modified by this transaction with the old version of the metadata in the log entry. After the recovery, PMA can continue to normally work.

The execution time of the recovery routine is determined by the size of the log region (scanning time) and the number of uncommitted transactions (recovering time). The size of the log region is fixed at 256KB in our implementation, which supports at most 1,024 threads to concurrently work, as discussed in Section III-F. Figure 12 shows the recovery time for different numbers of uncommitted transactions. When the number of uncommitted transactions is small, the scanning

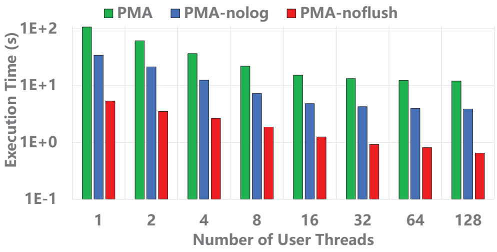


Fig. 11. Overheads of logs and flushes.

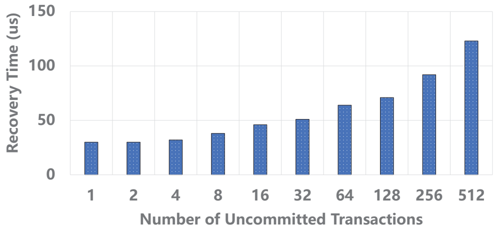


Fig. 12. The recovery performance of PMA.

time dominates (approximately 30 microseconds on average). However, when the number of uncommitted transactions grows, the recovery time linearly increases. According to our experiments, 512 uncommitted transactions can be recovered in approximately 120 microseconds.

V. Related Work

DRAM Allocators. *Hoard* [9] is a fast, highly scalable, and memory-efficient allocator that has been widely used in various platforms. This allocator achieves good scalability and avoids false sharing simultaneously. *Jemalloc* [11] is an allocator for FreeBSD to provide scalable concurrent allocation for multi-processor computer systems. *TCMalloc* [10] is a commercial allocator developed by Google, which employs thread-caching mechanism to provide high performance. *LR-Malloc* [12] is a modern lock-free allocator that leverages atomic instruction CAS to achieve high-efficiency concurrency control. These allocators are all designed for volatile memory.

Log-based PM Allocators. Persistent memory allocators need to consider the states of metadata (persisted or not), preventing metadata from corruption when a system failure occurs. Write-ahead log is employed in many PM allocators, such as *Poseidon* [13], *NVMalloc* [14], *nvm_malloc* [15], *PAllocator* [16] and *NVAlloc-log* [19]. Log-based allocators can roll back their metadata to the nearest consistent state after a crash.

GC-based PM Allocators. To reduce the overheads of writing logs and flushing metadata, some modern PM allocators, like *Makalu* [17], *Ralloc* [18] and *NVAlloc-GC* [19], adopt offline garbage collection (GC) to relax the online constraints of heap metadata persistence. They generate some root objects in the heap to trace all live user objects and traverse the *persistent heap* from the persistent root pointers during garbage collection. Any object that is unreachable from the root object will be reclaimed. They eliminate the overheads of write-ahead log but also introduce the overheads of GC.

VI. Conclusion

This paper elaborates the essential problems about persistent memory allocation/deallocation, including the corruption of metadata and the invalidity of virtual address pointers after a crash. To address these problems, we propose a persistent memory allocator (PMA) with high efficiency and crash consistency guarantee. PMA uses a two-level memory management strategy and reserves a private memory pool for each thread to achieve high concurrency. PMA designs a lightweight write-ahead log to protect the integrity and consistency of metadata and implements a lightweight persistent pointer to permanently reference a persistent memory object. We evaluate PMA on a real PM platform in terms of multiple metrics. Experimental results demonstrate that PMA can improve the throughput of allocation operations by up to 19.1x and reduce the tail latency by orders of magnitude under some certain workloads. The source code of PMA is available at https://github.com/HUSTxyxiang/PMA/.

References

[1] pmem.io. Persistent memory development kit. **https://pmem.io/pmdk/**.

[2] F. Yang, J. Kangy, S. Ma and J. Huai, “A Highly Non-Volatile Memory Scalable and Efficient File System,” in *ICCD*, 2018.

[3] J. Yang, J. Kim, M. Hoseinzadeh, J. Izraelevitz and S. Swanson, “An Empirical Guide to the Behavior and Use of Scalable Persistent Memory,” in *FAST*, 2020.

[4] H. Bae, M. Kwon, D. Gouk, S. Han, S. Koh, C. Lee, D. Park and M. Jung, “Empirical Guide to Use of Persistent Memory for Large-Scale In-Memory Graph Analysis,” in *ICCD*, 2021.

[5] M. Weiland, H. Brunst, T. Quintino, N. Johnson, O. Iffrig, S. D. Smart, C. Herold, A. Bonanni, A. Jackson and M. Parsons, “An early evaluation of Intel’s optane DC persistent memory module and its impact on high-performance scientific applications,” in *SC*, 2019.

[6] J. Xu and S. Swanson, “NOVA: A log-structured file system for hybrid volatile/non-volatile main memories,” in *FAST*, 2016.

[7] S. Zhong, C. Ye, G. Hu, S. Qu, A. Arpaci-Dusseau, R. Arpaci-Dusseau and M. Swift, “MadFS: Per-File Virtualization for Userspace Persistent Memory Filesystems,” in *FAST*, 2023.

[8] S. Liu, K. Seemakhupt, Y. Wei, T. Wenisch, A. Kolli and S. Khan, “Cross-Failure Bug Detection in Persistent Memory Programs,” in *ASPLOS*, 2020.

[9] E. D. Berger, K. S. McKinley, R. D. Blumofe and P. R. Wilson, “Hoard: A Scalable Memory Allocator for Multithreaded Applications,” in *ASPLOS*, 2000.

[10] Google. TCMalloc: Thread-Caching Malloc. **https://google.github.io/tcmalloc/**.

[11] J. Evans, “A Scalable Concurrent malloc(3) Implementation for FreeBSD,” in *BSDCan*, 2006.

[12] R. Leite and R. Rocha, “A Modern and Competitive Lock-Free Dynamic Memory Allocator,” in *VECPAR*, 2018.

[13] A. Demeri, W. H. Kim, R. M. Krishnan, J. Kim, M. Ismail and C. Min, “Poseidon: Safe, Fast and Scalable Persistent Memory Allocator,” in *Middleware*, 2020.

[14] I. Moraru, D. G. Andersen, M. Kaminsky, N. Tolia, P. Ranganathan and N. Binkert, “Consistent, durable, and safe memory management for byte-addressable non volatile main memory,” in *TRIOS*, 2013.

[15] D. Schwalb, T. Berning, M. Faust, M. Dreseler and H. Plattner, “nvm_alloc: Memory allocation for nvram,” in *VLDB*, 2015.

[16] I. Oukid, D. Booss, A. Lespinasse, W. Lehner, T. Willhalm and G. Gomes, “Memory Management Techniques for Large-Scale Persistent-Main-Memory Systems,” in *VLDB*, 2017.

[17] K. Bhandari, D. R. Chakrabarti and H. J. Boehm, “Makalu: Fast Recoverable Allocation of Non-volatile Memory,” in *OOPSLA*, 2016.

[18] W. Cai, H. Wen, H. A. Beadle, C. Kjellqvist, M. Hedayati and M. L. Scott, “Understanding and Optimizing Persistent Memory Allocation,” in *ISMM*, 2020.

[19] Z. Dang, S. He, P. Hong, Z. Li, X. Zhang, X. Sun and G. Chen, “NVAlloc: Rethinking Heap Metadata Management in Persistent Memory Allocators,” in *ASPLOS*, 2022.