Lewat: A Lightweight, Efficient and Wear-Aware Transactional Persistent Memory System

Kaixin Huang, Sumin Li, Linpeng Huang, *Member, IEEE,*
Kian-Lee Tan, *Senior Member, IEEE,* and Hong Mei, *Fellow, IEEE*

***Abstract*—Emerging non-volatile memory (also termed as persistent memory, PM) technologies promise persistence, byte-addressability and DRAM-like read/write latency. A proliferation of persistent memory systems have been proposed to leverage PM for fast data persistence and expose malloc-like persistent APIs. By eliminating disk I/Os, these systems gain low-latency and high-throughput access performance for persistent data. However, there still exist non-negligible limitations in these systems, such as frequent context switches, inefficient allocation, heavy logging overhead and lack of wear-leveling techniques. To solve these problems, we develop Lewat, a lightweight, efficient and wear-aware transactional persistent memory system. Lewat is built in user-layer to avoid kernel/user layer context switches and enables lightweight persistent data access. We decouple the data space into slot zone and page zone. Based on this, we design different allocators in these two zones to achieve efficient allocation performance for both small-sized data and large-sized data. To minimize logging overhead, we propose an efficient adaptive logging framework. The main idea is to utilize different logging techniques for different workloads. We also propose a suite of system-coupled wear-leveling techniques that contain wear-aware allocation, wear-aware update and write reduction. We evaluate Lewat on a real non-volatile memory platform and the experimental results show that compared with state-of-the-art persistent memory systems, Lewat has much lower latency and higher throughput.**

***Index Terms*—non-volatile memory, memory management, persistence, consistency, crash recovery, wear-leveling**

I. Introduction

Emerging non-volatile memory technologies, such as PCM [6], STT-RAM [7] and 3D XPoint [8], are expected to offer cheap yet fast storage for managing persistent data. These devices are often termed as persistent memory (PM), which retain both the byte-addressability of DRAM and the non-volatility of disk/SSD. In particular, recently-released Intel Optane DC Persistent Memory (also named AEP) is the first commercially-available non-volatile memory product [9].

To leverage fast data persistence of PM, a proliferation of researches focus on developing persistent memory systems that expose malloc-like APIs. With these systems, programmers can manipulate persistent data directly in allocated persistent memory space. For example, Mnemosyne [1] is developed as a persistent memory library to manage data with persistent pointers that are durable across system crash. Meanwhile, NV-Heaps [2] is a persistent object store for user-defined data structures, such as lists and trees. NV-Heaps utilizes specific namespaces to identify persistent objects. PMDK[1] [3] is developed based on a persistent memory file system [10], and organizes persistent space as a set of pools. Each pool is physically based on a persistent file in PMFS. All these three persistent memory systems employ mmaped file-based persistent memory implementation since they are mainly built as kernel modules. HEAPO [4] takes a further step by developing a native heap for fast user-layer memory allocation/recycling and data access. In our previous work [5], we developed EFLightPM that uses a hybrid kernel- and user-layer management mechanism for persistent data. It supports both lightweight space consumption for large-sized data and efficient access for small-sized data. All these persistent memory systems provide solutions for persistent data retrieval and data consistency guarantee. Table I summarizes the main design issues of these systems.

While these proposed systems facilitate persistent memory-based programming, they incur non-negligible bottlenecks that lead to dramatic performance overhead or limit their usage in real development environment. The bottlenecks can be summarized into four aspects, as discussed below.

1) Kernel/user layer context switch. Persistent memory systems such as Mnemosyne, NV-Heaps and PMDK, are implemented in a mmaped file-based method. They require at least two context switches each time the *mmap*-like syscall is used for accessing a persistent data region, which significantly decreases the system performance. To examine the context switch overhead caused by mmap sycalls, we compare the metadata operation latency between kernel-layer based implementation (PMDK) and user-layer based implementation (HEAPO). We observe that PMDK introduces 3.75x, 3.22x and 5.72x latency for allocate, recycle and attach operations, respectively, when compared with HEAPO. The results are consistent with that reported in [4], [11].

2) Inefficient allocation. The mainstream persistent memory systems employ buddy system-like mechanism for large-size allocations and slab-based mechanism for small-size allocations. However, allocation and recycling for small-sized data may suffer more complex memory splits and combinations due to fragmentations [12], which increase the total latency of executions. Allocation for large-sized data can also lead to low efficiency because of multiple-level buddy probing and poor scalability caused by buddy collisions.

3) Heavy logging overhead. To guarantee data consistency, existing systems employ either redo or undo logging techniques which incur the cost of double writes and ordering overhead in the critical path. We conduct an experiment in HEAPO by porting other systems' logging implementations into HEAPO's transaction framework. It is observed that for

[1]PMDK includes several libraries, and we use PMDK to refer to its *libpmemobj* library throughout this paper.

TABLE I
COMPARISON OF DIFFERENT PERSISTENT MEMORY SYSTEMS

Dimension	Mnemosyne [1]	NV-Heaps [2]	PMDK [3]	HEAPO [4]	EFLightPM [5]	Lewat
Implementation	mmaped file	mmaped file	mmaped file	user-layer heap	user-layer heap	user-layer heap
Logging Scheme	redo logging	undo logging	redo logging	undo logging	hybrid logging	adaptive logging
Data Resizing	×	×	✓	✓	×	✓
Allocator Optimization	×	×	×	×	×	✓
Concurrency Optimization	×	×	×	×	✓	✓
Multiple-Region Transaction	✓	×	✓	×	×	✓
Wear-Leveling	×	×	×	×	×	✓

small data regions (e.g., 64 bytes), the log writes account for nearly 57% and 48% overall overhead in undo logging and redo logging, respectively. For large-sized data (e.g., 2KB), the proportions of logging writes can even reach 74% and 63% for the two logging techniques. The reason is that logging writes incur both double write overhead and extra persistence overhead caused by clflush/mfence [13].

4) Lack of wear-leveling. A PM cell will wear out after a certain number of writes. For example, a PCM cell typically fails permanently after 10^7 to 10^8 writes [14]. Poorly designed applications or malicious programs on current persistent memory systems may break PM down in minutes [15]. Although it is reported that the latest AEP product is self-equipped with Intel's hardware-layer wear-leveling techniques [16], techniques for wear-leveling should still be considered because the hardware-layer data migration may significantly decrease the write performance to the same data addresses, as indicated by previous work [17]. With proper software-layer wear-leveling, such performance hit can be avoided.

To address these bottlenecks mentioned above, we propose Lewat, a lightweight, efficient and wear-aware transactional persistent memory system. To avoid heavy context switch overhead for persistent region access, we expose the entire persistent memory space in user layer as a native heap. The data space in Lewat is divided into slot zone and data zone. To enable efficient allocation for small-sized (e.g., a few bytes) data, we design a special page data structure called slotted page (in slot zone), which includes multiple slots. A per-core slot allocation index is built in DRAM space to efficiently allocate memory in slot units. To enable efficient allocation for large-sized (e.g., KBs, MBs or even GBs) data, we design a layered bitmap to manage the page zone. Page zone allows atomic allocation in page-level, chunk-level and group-level, where each chunk contains a set of contiguous pages and each group contains a set of contiguous chunks. For scalability, we partition chunks to different cores, hence multiple allocation threads can be parallelized in chunk level. To reduce the heavy double write overhead and ordering overhead caused by transaction logging, we propose an adaptive logging framework. The main idea is to dynamically utilize different logging techniques for different write workloads to minimize the write overhead and ordering overhead in the critical path. The main contributions of this paper are summarized as follows.

• We develop a novel persistent memory system, Lewat. To the best of our knowledge, we are the first to incorporate all these functionalities, including support for small and large sized allocation, transaction mechanism, concurrency control, wear-leveling, and crash recovery in a single persistent memory system.

• We design different persistent memory allocators for slot zone and page zone, respectively. These can offer efficient and scalable allocation performance for both small-sized data and large-sized data.

• We propose an efficient adaptive logging framework to dynamically adjust the logging technique to different write workloads. In this way, the write overhead for data consistency guarantee is minimized for each type of workload.

• We design a suite of system-coupled concurrency mechanisms and wear-leveling techniques. They fully exploit the performance potential of Lewat by flexibly utilizing the features of adaptive logging framework.

• We conduct an in-depth evaluation of Lewat using both synthetic benchmarks and industrial workloads on a real PM platform. The results show that compared with existing systems, Lewat reduces the latency of metadata access by 8.6%-91.7% and improves the data access performance to 3.3x-12.9x for various operations. As for transaction throughput, the speedup of Lewat is up to 4.02x for YCSB workloads and 6.10x for Facebook Memcached workloads.

Organization. Section II introduces the background of this work. Section III provides the design of Lewat and Section IV elaborates Lewat's adaptive logging framework. In section V, we present the wear-leveling mechanism of Lewat. Section VI describes the crash recovery algorithm and Section VII presents results of an experimental study. We summarize related work in Section VIII and conclude this paper in Section IX.

II. BACKGROUND

A. Persistent Memory

Emerging non-volatile memory (also termed as persistent memory, PM) technologies are closing the gap of performance, cost-per-bit, and capacity between low-latency, volatile memory technologies (e.g. DRAM) and high-capacity, persistent storage technologies (e.g. disk, SSD, Flash) [6]–[8], [18]. Next-generation PM technologies (such as PCM [6], STT-RAM [7] and 3D XPoint [8]) are byte-addressable and projections show that their performance may approach that of DRAM [6], [7], [18]. In particular, Intel Optane DC Persistent Memory (also named Apache Pass, AEP) is the first commercially-available non-volatile memory product [9]. Attaching PMs to the main memory bus provides a raw storage

medium that can be orders of magnitude faster than modern persistent storage medium such as disk and SSD. PM also presents a few new technical challenges and has inspired a host of research projects on topics including OS management of PM [19], specialized PM-based file systems [10], [20], and persistent memory systems [1]–[5], [21]. This work focuses on persistent memory system, which can support efficient data persistence for many storage applications, such as Memcached [22], Redis [23] and Berkely DB [24].

B. Review of Existing Persistent Memory Systems

In Table I, we compare several persistent memory systems that exploit the non-volatility and byte-addressability of PM. Mnemosyne [1] and EFLightPM [5] are persistent memory systems that focus on transactional updates of both small-sized and large-sized data. Other persistent memory systems, such as NV-heaps [2], HEAPO [4] and PMDK [3] are mainly designed to serve large-sized objects. Mnemosyne, NV-Heaps and PMDK adopt memory-mapped file-based implementation to access persistent data, which causes frequent context switches between kernel layer and user layer. HEAPO and EFLightPM avoid the overhead of multiple *mmap* syscalls by managing persistent memory space in user-layer. However, for free space allocation and recycling, they simply utilize the buddy system algorithm and slab-based design, which may incur inefficient allocation for small-sized data or large-sized data due to allocator metadata persistence and bucket split/merge management.

Data consistency is a significant issue in persistent memory. Current processors support 8 byte atomic writes natively. For a large object that is more than 8 bytes, when there is a system crash, it is possible that only a portion of the whole data object may be persisted in PM, leading to inconsistent partial writes. To make matter worse, current CPU and caches may reorder data writes to memory, without respect to program order [25]. For non-exchangeable data updates in a transaction, wrong order may occur when a crash happens, which leads to inconsistency. Instruction primitives such as `clflush/clwb/ntstore` and `mfence/sfence` are used in PM-based systems to preserve ordering writes [13], [25], [26]. Existing persistent memory systems utilize logging techniques [27] to guarantee data consistency (e.g., redo logging in Mnemosyne and PMDK, undo logging in HEAPO and NV-Heaps). However, a monotone logging scheme can lead to heavy double write overhead and persistence ordering overhead [5], [28].

As shown in Table I, our proposed Lewat is a comprehensive and full-fledged PM memory system that incorporates all critical features for practical use. We shall discuss Lewat in the next few sections.

III. LEWAT

To address the bottlenecks in existing persistent memory systems, we design and implement Lewat, a lightweight, efficient and wear-aware transactional system to manage persistent data in PM. In this section, we will first introduce the APIs and storage organization of Lewat (III-A). Then we will describe the space management strategy for persistent memory (III-B) and finally elaborate the allocation/recycling mechanism (III-C). We defer the discussions on Lewat's adaptive logging scheme, concurrency control, crash recovery and wear-leveling mechanisms to subsequent sections.

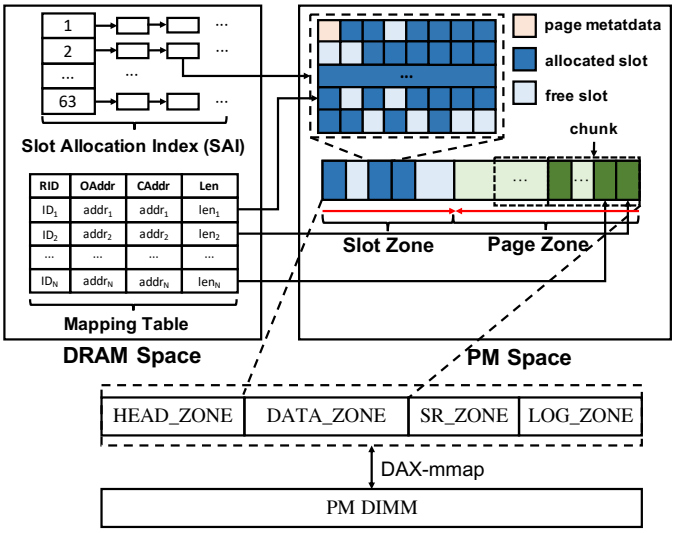


Fig. 1. Storage Organization of Lewat.

A. Lewat APIs and Storage Organization

Lewat provides a suite of APIs for users, as shown in Table II, which can be classified into three categories.

- Metadata Access API: *p_malloc*, *p_free* and *p_attach*. As their names imply, *p_malloc*/*p_free* are analogous to malloc/free in the standard C library. *p_attach* is used to obtain the address of persistent data. When these APIs are called, metadata read or writes are required.
- Data Access API: *p_get*, *p_write*, *p_append* and *p_truncate*. *p_get* and *p_write* are utilized to read and write persistent data, respectively. As for *p_append* and *p_truncate*, they should be used when append-only updates and data truncation are needed.
- Transaction API: *stx_start*, *stx_end*, *stx_abort*, *mtx_start*, *mtx_end*, and *mtx_abort*. Among them, the first three APIs are sufficient for a single-region transaction. As for multiple-region transactions, the latter three APIs should be properly applied as well.

Notice that for all Lewat APIs, region ID (RID) is a significant parameter because it is used to identify a unique persistent region in Lewat. Lewat APIs are self-explanatory and straightforward to use for building high-level applications such as KV-Stores and DBMSs atop of it. The storage organization of Lewat is given in Figure 1. Persistent data are stored in PM and we build indexes in DRAM. We divide the entire persistent data area (i.e., DATA_ZONE) into two parts: slot zone for storing small region and page zone for storing large regions. Although both zones share the same page size of 4KB, the slot zone has much smaller basic storage/allocation unit (i.e., slot). A slotted page can contain multiple slots, with the first slot holding metadata for the whole page. Suppose

TABLE II
LEWAT APIS

Category	API	Input	Output	Description
Metadata Access	*p_malloc*	*RID, size*	*addr*	allocate space for specified RID with specified size
	p_free	*RID*	*NONE*	recycle a specified region
	p_attach	*RID*	*addr*	get address of a specified persistent region
Data Access	*p_get*	*RID, rbuffer*	*rbuffer*	read data out from specified persistent region
	p_write	*RID, wbuffer, addr, len*	*NONE*	write data to a specified address with given length
	p_append	*RID, wbuffer, len*	0 or -1	append data to a specified address with given length
	p_truncate	*RID, len*	0 or -1	truncate data of specified address with given length
Transaction	*stx_start*	*RID(, flag)*	0 or -1	start a single-region transaction or join in an existing multi-region transaction with specified/adaptive logging mode for current region
	stx_end	*RID*	0 or -1	end a single-region transaction
	stx_abort	*RID*	*NONE*	abort a single-region transaction
	mtx_start	*NONE*	*TID*	start a multiple-region transaction
	mtx_end	*TID*	0 or -1	end a multiple-region transaction
	mtx_abort	*TID*	*NONE*	abort a multiple-region transaction

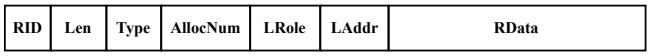
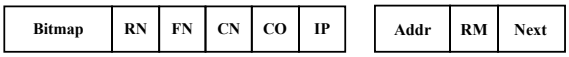


Fig. 2. Structure of persistent region.



(a) Slotted page metadata (b) SAI pointer node

Fig. 3. Organization of metadata and SAI for slot zone allocation.

that a slot is 64 bytes[2] in size, then there are 64 slots in a slotted page, as shown in Figure 1. A DRAM-resided slot allocation index (SAI), which is made up of multiple linked-list buckets, manage slotted pages with different orders (i.e., maximum number of consecutive free slots).

In page zone, page is the basic storage unit for persistent data. Multiple consecutive pages are organized into chunks. Concretely, a chunk can contain 64 consecutive pages, with total space size of 256 KB. Similarly, consecutive chunks can also constitute a group, whose size is 16 MB. In this way, Lewat can support chunk-level and group-level allocation/recycling for large-sized data. Furthermore, it also contributes to high concurrency for multiple threads by allocating/recycling in different chunks and groups.

Persistent data objects in Lewat are packed as persistent regions by adding corresponding metadata. A mapping table is built in DRAM to store the actual storage locations (i.e., start address of data area in the slots or pages - this is the O_Addr field in the figure; we will discuss how C_Addr is used in the COUL logging scheme and the wear-leveling scheme) and data length for valid region IDs. We implement the mapping table with bucket-based k-cuckoo hashing where k is 2.

B. Persistent Memory Management

As shown in Figure 1, the entire persistent memory space of Lewat includes four partitions: 1) the HEAD_ZONE, which stores system metadata of Lewat, includes system-level information such as free space size, valid region numbers, and etc; 2) the DATA_ZONE, which includes slot zone and page zone, has already been presented in III-A; 3) the SR_ZONE and 4) LOG_ZONE, which are used to guarantee transactional writes.

As introduced above, Lewat manages persistent data objects as persistent regions and each persistent region has a unique identifier, called RID. Figure 2 shows how a persistent region is structured. It can be divided into three parts. (1) Metadata part: *RID* (region ID), *Len* (data length), *Type* (slot or page) and *AllocNum* (number of allocated slots or pages). (2) Data part: *RData* (user-required persistent data); (3) Log part: *LRole* (logging mode) and *LAddr* (log block offset). For persistent region in slot zone, metadata and log entry consume only 19 bytes (8 bytes for *RID*, 2 bytes for *Len*, 1 byte for *Type* and *AllocNum*, 8 bytes for *LRole* and *LAddr*); while in page zone, they consume 28 bytes (the differences are: 8 bytes for *Len*, 4 bytes for *Type* and *AllocNum*). When a persistent region is newly created or recycled, an entry in the mapping table should be created or cleared correspondingly.

A natural concern for the zone division of DATA_ZONE is that severe space resource wastage may occur for certain workloads. For example, when an Lewat-based application serves an allocation-heavy workload for small-sized data, free space in the page zone cannot be used effectively. We address this issue by devising a dynamic border-moving strategy for the two zones. Space allocations in Lewat comply with the following rule: slotted pages in the slot zone are allocated from low address to high address, while normal pages in the page zone are allocated in the reverse order. When either type of pages are insufficient, Lewat will transform the other type of free pages which are adjacent to the border, to the required type of pages. In Lewat, the smallest unit of transformation is a chunk, which contains 64 pages. When border-adjacent pages are not free, an extra data migration procedure is needed. Note that the border-related information such as slot zone size and page zone size is managed in the HEAD_ZONE.

C. Persistent Memory Allocation & Recycling

In Lewat, we design different allocators for slot zone and page zone. Figure 3 shows the details of slotted page metadata and pointer node in slot allocation index. As illustrated in

[2]the slot size is configurable in Lewat. By default, it is set to 64 bytes.

Figure 3(a), *Bitmap* is used to manage allocated and free slots by setting 1s and 0s, respectively. *RN* records the number of valid regions while *FN* stores the number of free slots. *CN* indicates the maximum number of contiguous free slots and *CS* records the start offset of such slots. *IP* is a reverse pointer that corresponds to a pointer node in SAI, for the ease of memory recycling. Although it points to a volatile space from persistent space, there is no data consistency issue here because Lewat will re-initiate its value during each system reboot. Figure 3(b) shows that a pointer node only includes three fields. *Addr* is an address pointing to a slotted page with a specified order. *RM*, short for round-robin mark, is a slot offset used for wear-aware allocation in one slotted page. *Next* points to the next index node in the same linked-list bucket.

For instance, if a persistent region requests 80 bytes for allocation, which is larger than one slot but smaller than two slots, then the allocating procedure works as follows.

(1) Locate the SAI bucket with order 2 and check the number of available pointer nodes belonging to this bucket.

(2) If the number is larger than 0, get the head pointer node and skip to the slotted page according to the *Addr* field. Otherwise, increase the order by 1 and restart step (1).

(3) Try to allocate 2 contiguous slots in the selected slotted page according to the *RM* field of the index node. If such an action fails, just allocate two contiguous slots according to the *CS* field of the slotted page metadata.

(4) If needed, update the *RM* field of the pointer node, and *Bitmap*, *RN*, *CN*, *CS* fields of the corresponding slotted page metadata.

(5) Move the SAI pointer node into a lower-order bucket if needed and update the system metadata in the Header area.

Lewat slot allocator is inspired by WAFA [29], but there are three significant differences between them. First, we do not use pointers to link each slotted page since *prev/next* pointers not only consume extra storage space in NVM, but also increase persistence overhead for pointer modifications. Furthermore, the page management and crash recovery can be more complex. In contrast, Lewat keeps the slotted page physically contiguous without extra management overhead and regards SAI as a reconstructable index structure in DRAM. Second, we optimize concurrent allocations for small-sized regions by allowing per-core slot zone allocator configuration. Concretely, each allocation thread is bound to a specified CPU core and is responsible for a range of slotted pages. Also, each allocation thread can maintain its own SAI. Third, the size of slot zone can dynamically grow or shrink depending on the needs of the application, adding more flexibility.

In the page zone, the basic allocation unit is page. In Lewat, we design a layered bitmap to manage the allocation status of each page, chunk and group. We also keep a layered lockmap in DRAM to both support fast allocations for large-sized memory requests and resolve concurrent allocation collisions. Figure 4 illustrates these two components, each with three layers. Each bit in the lowest page layer corresponds to a single page. Note that a higher-layer bit in both bitmap and lockmap summarizes the information of 64 lower-layer bits. The difference is that in the layered bitmap, a higher-layer bit is 1 only when all its 64 lower-layer bits are 1s, indicating a full chunk or group; while in the layered lockmap, a higher-layer bit is 1 as long as at least one of its lower-layer bits is 1, meaning that a full chunk/group allocation corresponding to current bit is unavailable. Some other works [3], [30] use tree index structures to manage large-block allocations. But they introduce extra in-NVM leaf-node pointer chasing overhead. To improve the scalability of page zone allocator, we partition chunks to different cores and only a group-level allocation request will collide for the global lock.

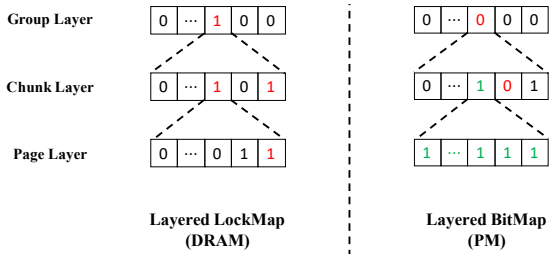


Fig. 4. Layered bitmap and lockmap.

In Lewat, the border size for allocation between small-sized data and large-sized data is set to 1/2 page size (i.e., 2 KB). Suppose that a *p_malloc* call requires 3KB persistent memory, although the slot zone allocator can serve such request, free space will not be allocated from the slot zone. Instead, Lewat selects a free page from the page zone for this request. In this way, Lewat allows a persistent region to obtain more space than it requests on allocation, thus efficient append operation can be conducted opportunistically without extra allocation overhead. As for truncate operation, Lewat adopts a lazy-recycling method, which only updates the *Len* field of persistent region in the critical path. Free space in a persistent region will be withdrawn when the available space resource is scarce (i.e., less than 10% of total size).

To better position the differences over existing PM allocators, we compare Lewat allocator against several state-of-the-art PM allocator proposals [29]–[32]. Our experimental study shows that Lewat allocator achieves the best throughput (per thread) compared to existing PM allocators for all different allocation sizes. The details are provided in Appendix A.

IV. Adaptive Logging

To guarantee crash consistency, conventional logging techniques such as redo logging and undo logging can be directly applied. However, these basic schemes are inefficient due to double writes and persistence ordering overhead. Some researchers have proposed optimized mechanisms for redo/undo logging in either software-layer [33]–[38] or hardware-layer [39]–[42]. However, it is inadequate to simply deploy one single logging technique as no method is superior for different types of workloads or can adapt to changing workloads [28]. In this paper, we propose a novel adaptive logging framework to dynamically utilize several state-of-the-art logging techniques for different workloads. The aim is to pick the most appropriate method that can minimize the write overhead for each type of workload.

We first distinguish workloads into four basic types: slot-write workload that writes or modifies a persistent region

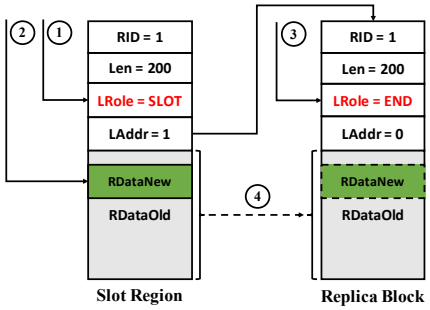


Fig. 5. Procedure of replica-enabled write-behind logging.

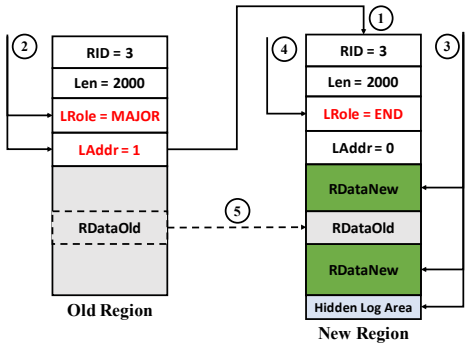


Fig. 6. Procedure of COW-based undo logging.

which can fit in a single slot, write-after-allocate workload that firstly writes a newly-created persistent region, major-update workload that updates all or a large portion of a persistent region, minor-update workload that updates only a small portion of a persistent region. In particular, the border between minor-update workload and major-update workload is half of the valid data size (i.e., 1/2 *Len* of a persistent region). Then we introduce various logging techniques to deal with different workloads (IV-A to IV-D). Finally, we integrate these logging techniques into our adaptive logging framework (IV-E). We also discuss concurrent access mechanism (IV-F) and multiple-region transaction (IV-G).

A. REWBL: Replica-Enabled Write-Behind Logging

The main idea of REWBL[3] is adapted from a previous transaction mechanism, Kamino-Tx [36]. Concretely, we keep a replica block for a slot region (short for slot-sized persistent region) and perform in-place update to the slot region directly. The SR_ZONE in Figure 1 is divided into fixed number of replica blocks, each of which is the same size of a slot. When a slot region is created after calling *p_malloc*, a replica block in the SR_ZONE is automatically allocated for it and they are related with each other via the *LAddr* field, which is a page offset of the replica block. Figure 5 shows the concept of REWBL, which employs a frontend-backend coordination model similar to EFLightPM [5]. The frontend procedure of REWBL has three steps: (1) initialize the *LRole* field to `SLOT`, marking a pending transaction for current slot region; (2) make in-place update; (3) set the *LRole* field of replica block to `END`, indicating a committed transaction. A transaction can immediately end after completing the frontend procedure. The backend procedure is responsible for the remaining tasks. Step (4) in Figure 5 copies the *RData* content of the entire slot to the replica block. The last step of backend procedure, which is not shown in Figure 5, is to reset both *LRole* fields to `NONE`. Since the SR_ZONE is reserved as fixed size, with the number of slot region increasing, there may be not enough free replica blocks. Kamino-Tx solves this problem by replacing least recently updated regions with those which are about to get updated and do not have a copy. Since most application working set sizes are skewed and contain a small percentage of the entire data set, this mechanism can be efficient and we also adopt this design in Lewat. Therefore, the space overhead of replica blocks for REWBL is under control.

The reason why REWBL supports in-place update is that a backup data copy already exists in a corresponding replica block, which acts as an undo log. Any inconsistent data caused during in-place update can be rolled back to a consistent state through scanning the replica block. REWBL eliminates not only the double write overhead in the critical path, but also minimizes the metadata overhead such as log block allocation and initiation. Therefore, it is expected to achieve high-performance write transactions on slot regions. Different from Kamino-Tx, we do not keep replica blocks for persistent regions that are larger than a slot in size. Kamino-Tx may consume a lot of backend write bandwidth for large-sized data writes, which can become a bottleneck of the overall throughput. REWBL does not suffer from such an issue because the granularity of replica block is fixed in slot size and cache-friendly for writes.

B. LOFU: Log-Free Update

The idea of LOFU is useful for write-after-allocate workload, which inherently does not require double write because there is no old data to maintain. A previous work on persistent B+-Tree [44] proposes this technique for insert operation in a tree node. In this paper, we extend its use to support efficient insert operation for all kinds of persistent regions. LOFU allows new writes to be directly persisted in the *RData* field. The transaction state is also recognized by the *LRole* field. In the beginning, the *LRole* state should transit from `ALLOC` to `FIRST`, indicating there is a pending write for a newly-allocated persistent region. After all writes are stored in the persistent region with in-place update, the *LRole* field can be reset to `NONE`.

Compared with those systems that do not differentiate write-after-allocate workload and other workloads, Lewat is expected to gain much lower latency and higher throughput when insert operations are heavy.

C. COUL: COW-Based Undo Logging

For major-update workload, we adopt the copy-on-write (COW) technique to decrease the double write overhead in

[3]REWBL is totally different from the write-behind logging mechanism proposed in [43], which utilizes append-only writes for database records (though they share the same "write-behind logging" term).

redo/undo logging. COW has been widely-used for pointer-based data structures [20], [37], [45]–[47]. In this paper, we extend its use for general persistent regions. COUL takes the old persistent region as a natural undo log.

The execution procedure is illustrated in Figure 6. Similar to REWBL, there are both frontend and backend procedures for COUL. The frontend procedure of COUL contains four steps: (1) allocate a new persistent region for absorbing new writes; (2) set the *LRole* field to `MAJOR` and *LAddr* field to the page offset of the new region; (3) write updates to the *RData* field of the new region while maintaining lightweight log records of updated addresses. (4) set the *LRole* field of new region to `END`, marking a committed transaction. Notice that the address information in step (3) is stored in a hidden log area of the new region, which obtains more memory space on allocation for this task. The backend procedure of COUL has two steps: (5) copy the unmodified data from old region to the new region, based on the records of updated address information stored in the hidden log area; (6) reset the *LRole* field of new region to `NONE` and recycle the space of old region. Notice that a corresponding item in the DRAM-resided mapping table (in Figure 1) should be atomically updated in accordance with the new region's address. Concretely, the C_Addr field (i.e., current address) should be updated to the new region's address while the O_Addr field (i.e., original address) should keep unmodified for future read/write indirection. For applications, the change of region's address in the mapping table is transparent and future access to the data address obtained by *p_attach* will be indirected to the new region with a simple calculation: target_addr = (access_addr - O_Addr) + C_Addr. Such indirection trick is similar to PTL [48], which also builds an virtual address mapping table with both source address and redirected address.

COUL is efficient for major-update workload in two aspects. First, it eliminates double writes in the critical path since the frontend procedure only writes to the newly-allocated region. The overhead generated by the new region allocation using COUL is less than 5% for 2KB update. With larger writes, such overhead can be even lower. Second, it reduces backend bandwidth consumption (compared with redo/undo logging) by copying less data to the region. A more interesting point about COUL is that it is critical to the implementation of wear-aware update, which will be discussed in section V.

D. CERL: Checksum-Enabled Redo Logging

For minor-update workload, we do not choose COUL because both the persistence ordering overhead in the frontend procedure and backgroud write overhead in the backend procedure can be heavy, thus affecting the overall system performance. Instead, we utilize CERL, which writes the updates to a log block first and then write updates back to the persistent region based on the log items. However, unlike the conventional redo logging method that may incur severe persistence ordering overhead, CERL is able to eliminate the uses of *clwb* and *store* when the updated size for a region is small. CERL is adapted from Pilaf's checksum-based verification mechanism [49], which is used to detect the write

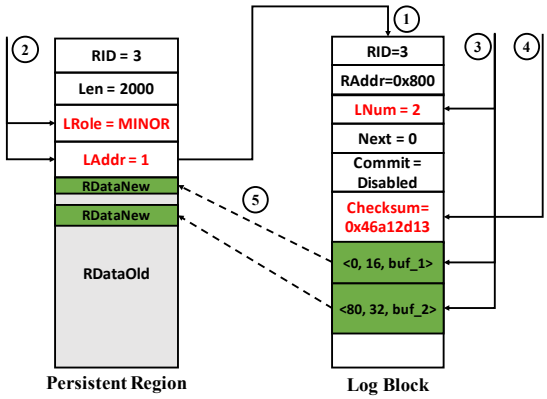


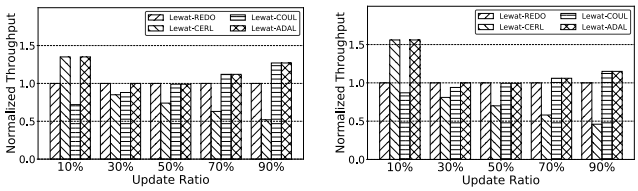
Fig. 7. Procedure of checksum-enabled redo logging.

completion of a key-value item for cross-network concurrency control. In this paper, similar to some other works [5], [21], [38], we employ the checksum for data consistency check during crash recovery.

Similar to a persistent region, a log block used in CERL also has two parts: metadata part and data part. The metadata part contains *RID*, *RAddr* (original region address), *LNum* (number of log records), *Next* (an offset of next log block), *Commit* (commit mark for current log block) and *Checksum* (a value computed through CRC64 on log records). The data part maintains several *(addr, len, value)* tuples to record new updates for the corresponding persistent region.

Figure 7 presents the procedure of CERL. The frontend procedure contains four steps: (1) allocate a persistent log and set log metadata; (2) initialize the *LRole* field to `MINOR` and *LAddr* field to the offset of the log block; (3) transform each write as a log appending operation in the persistent log and update the *LNum* field correspondingly; (4) compute the checksum of all log records (excluding log metadata). There are two steps in the backend procedure: (5) persist the log and copy updates back to the *RData* field of original persistent region; (6) reset the *LRole* field of the persistent region to `NONE` and recycle the space of the log block. Notice that there is no need to maintain the persistence order in the frontend procedure when *Checksum* field is used. The loose-ordering write design of CERL is inspired by the nature of checksum: a correct checksum in the persistent log indicates an integrated transaction. Any inconsistent data in the log will produce a different checksum. During recovery, if the recalculated checksum matches the *Checksum* field, the log content is considered intact. We adopt CRC64 to implement CERL and the averaged computing overhead for each byte is only several CPU clocks [50]. When the update size is small, CERL can remarkably outperform traditional logging techniques. For instance, compared with redo logging used in PMDK, CERL reduced over 25% and 15% latency overhead for update size in 16 bytes and 64 bytes, respectively.

However, for a large-size persistent region, a minor-update transaction may still update hundreds or thousands bytes of data. Under such situation, checksum-based mechanism can be inefficient due to the heavy computation cost. Fortunately, *Commit* field of persistent log can be utilized to figure out this



(a) Skewed writes (b) Uniform writes

Fig. 8. Throughput comparison of different logging techniques for 2KB-region updates.

issue. When the updated size of a minor-update transaction is too large, CERL will no longer compute the checksum of log records. Instead, it enables the *Commit* field by marking it as `True` after all log records are persisted in the frontend. The backend procedure is similar except that the task of persistence is skipped. Actually, this method follows the same logic of conventional asynchronous redo logging. Therefore, to distinguish these two logging techniques of using log blocks, we call the second one REDO in the following content.

E. Adaptive Logging Framework

As none of the logging techniques (redo/undo logging and the variants) performs well under a diversity of (or changing) workloads, we propose an adaptive logging framework that dynamically picks the logging method that best fits the real-time workload at that instant. To the best of our knowledge, we are the first to propose a comprehensive adaptive logging framework that integrate multiple state-of-the-art logging techniques to support efficient writes for different workloads in a PM-based system.

The challenge lies in recognizing real-time workload type on specified persistent region. In Lewat-based programming, when calling *stx_start*, slot-write workload and write-after-allocate workload can be identified directly. However, it is not possible to distinguish between major-update and minor-update workload at *stx_start* call since we do not have information on the total update size. Only when the *stx_end* interface is called, a transaction is considered complete and the total update size for a persistent region is acknowledged. The conflict is that without using concrete logging technique based on the correct workload type, *p_write* will not work properly to update data.

To address this problem, we defer the decision of workload type beween major-update and minor-update to the moment of first data write. That is, if the workload type cannot be inferred from the execution of *stx_start*, then Lewat can set a logging mode in the *LRole* field after the first *p_write* is called. Concretely, if the update size of the first write is smaller than half of the total data size, Lewat would set *LRole* field to `MINOR`. Otherwise, it would set *LRole* field to `MAJOR`.

With such compromise, Lewat can fulfill adaptive logging by checking the following policies in strict order.

(1) On calling *stx_start*, if the size of this persistent region can fit in a single slot, set the *LRole* field as `SLOT` and utilize REWBL for slot-write workload; otherwise check policy (2).

(2) If the *LRole* field is `ALLOC`, set *LRole* as `FIRST` and utilize LOFU for write-after-allocate workload; otherwise check policy (3).

(3) Set *LRole* as `PENDING` and wait for the first write to occur. On calling *p_write*, if the write size is smaller than half of the total data size, set *LRole* as `MINOR` and check policy (4); Otherwise set *LRole* as `MAJOR` and utilize COUL for major-update workload.

(4) On calling *stx_end*, count the total size of updates. If the total update size is smaller than or equal to 512 bytes[4], utilize CERL for minor-update workload; otherwise utilize REDO for minor-update workload.

Notice that normally we use *clwb* and *sfence* to guarantee the persistence ordering for PM writes in Lewat. However, it is reported that *ntstore* can achieve better write throughput for large writes [16], [26]. Therefore, we particularly utilize *ntstore* for COUL technique, which serves major-update workload that always lead to large writes.

Figure 8 compares the throughput performance of using different logging techniques in Lewat for both skewed and uniform write-only workloads. As REWBL and LOFU handle special cases (i.e., small sized data and insertions into newly allocated regions), our adaptive framework would also select them as the logging scheme to be applied in these cases. Hence we do not discuss them in our study. Instead, we focus on just COUL, CERL and REDO. Notice that skewed writes are zipfian-distributed with skewness parameter s=0.99. Lewat-ADAL indicates the default Lewat configuration that uses adaptive logging. For skewed writes, as shown in Figure 8(a), CERL gains 1.35x and 1.88x speedup for 10% updates, compared with REDO and COUL, respectively. Note that the checksum calculation for small data size is quite efficient. COUL requires to migrate more data than REDO and CERL in the backend procedure for minor-update workload. When it comes to major-update workload, COUL outperforms REDO and CERL by 1.12x-1.27x and 1.77x-2.54x, respectively. This is because for large size of data updates, the calculation cost in CERL can be heavy and there are more background writes in REDO than COUL. Notice that skewed writes may cause frequent write collisions on the same persistent regions. Hence, the backend procedure which exists in all the compared logging techniques, may become the performance bottleneck. As for uniform writes, CERL achieves much higher performance speedup (i.e., 1.5x) compared with REDO, while the overhead of COUL becomes lower (i.e., less than 13%) for the minor-update workload. The reason is that updates are evenly distributed across different persistent regions with uniform writes, and hence write collisions on the same regions become much fewer. As a result, the backend procedure for REDO, CERL and COUL is no longer a serious bottleneck. On the contrary, the throughput is highly dependent on the frontend execution efficiency, where REDO and COUL are at par with each other. In conclusion, all these three logging techniques (i.e., CERL, REDO and COUL) in our adaptive logging framework are critical for performance speedup.

[4]In our study, we find that CERL performs well for update size of no more than 512 bytes.

Although there are many existing works for optimizing redo logging, undo logging, redo-undo logging, we argue that they cannot fit for all kinds of workloads. A comparison for the software-based transaction mechanisms with Lewat's adaptive logging is provided in Table VI in Appendix B.

F. Concurrent Access Mechanism

To further increase the throughput, Lewat also supports concurrent accesses to persistent regions. Here, we present our concurrent access mechanism that guarantees data consistency for multi-threading environment on persistent memory systems. Generally, there are four types of concurrent access collision: (1) read-in-read, which reads a region that is being read by another thread; (2) read-in-write, which reads a region that is being updated by another thread; (3) write-in-read, which updates a region that is being read by another thread and (4) write-in-write, which updates a region that is being modified by another thread.

Now, clearly, read-in-read collision will not result in any data inconsistency, thus we focus on handling read/write collision and write/write collision. For example, a read-in-write operation may obtain partially-updated data from a region while a write-in-write operation may mix up updates from two transactions to the same region. To figure out such problems, we employ a lock-based concurrent access mechanism. There is a DRAM-resided 8-byte read-write locker for each persistent region, with 4 bytes representing read lock field (RL) and the other 4 bytes representing write lock field (WL). The update on the read-write locker can be performed atomically thus it is lightweight to acquire and release locks. Actually, RL corresponding to a region is a counter for the total number of active read operations on the region, and acquiring a read lock is an atomic fetch-and-add operation in nature. As for WL, it is merely a write indicator with finite states. Acquiring a write lock is actually an atomic compare-and-swap operation.

In Lewat, we adopt an exclusive write approach (i.e., at most one write lock for a region) so that write-in-write collisions cannot happen. To eliminate write/write collisions, Lewat utilizes a per-core RID-based write processing algorithm. That is, different write transactions to the same persistent region will be allocated to a specified CPU core and hence they can be processed sequentially in one core. Normally, a read lock can be acquired regardless of the lock status of the read-write locker. Therefore, read-in-write operation can be well supported. Write-in-read operation is a little more complicated. An update may interfere with the pending read procedure if the write/read addresses are overlapped. As such, a natural restriction is that a write lock on a region can only be acquired when there is no active read operations on the region. However, a write operation may have to endlessly wait for the release of all read locks under read-heavy workload. To avoid such starvation issue, we introduce a write registering technique. That is, when the waiting time is longer than a preset threshold, a write task can "register" on the write-lock to block potential future read locks. Therefore, when the remaining read jobs are finished and all read locks are released, the write operation can be processed favorably. On the other hand, when the write addresses and ongoing read addresses do not overlap, write-in-read operations can be safely executed to improve system throughput.

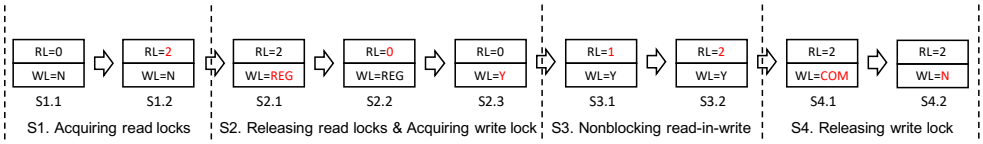
Based on the above analysis, we design different concurrent access mechanisms for slot regions (i.e., persistent regions that can fit in one slot) and normal regions (i.e., all persistent regions except slot regions). Figure 9 shows two possible sequences of read-write locker modifications for reading/writing the two types of regions, respectively. In Figure 9(a), there are multiple concurrent reads in stage S1 and RL records the number of read locks on the current region. During this procedure, data is directly read out from the *RData* field of the persistent region. A write transaction acquires the write lock by "registering" on WL to temporarily block future reads and waits for the release of all existing read locks. The reason why slot region does not support write-in-read operation is that REWBL technique (as introduced in section IV-A) always makes in-place updates first, which may affect the correctness of concurrent read operations. In stage S3, read-in-write operations are allowed and the read requests will first read the data from the replica block of the current slot region and then verify the value of WL. If WL is not changed after this read, then the updates in the slot region have not been copied to its replica block yet, and hence reading from the replica block is a safe operation. Otherwise, data should be read from the slot region again if (1) transaction is committed (S4.1) or (2) write lock is released (S4.2).

Unlike slot regions, logging techniques including COUL, CERL and REDO for normal regions do not perform in-place updates at first. Exploiting this point, write-in-read operations are supported by allowing write lock to be acquired concurrently with multiple read locks, as S2 in Figure 9(b) shows. However, when updates are copied back from the log block due to the backend procedure of CERL and REDO technique, the ongoing read operations can no longer be safe. Similar to the method of acquiring a write lock for slot region, the write operation for normal region will "register" on WL to block future reads and wait for the release of all existing read locks, as shown in S3. Afterwards, the write lock state can be updated to "committed" and updates will be copied back to the original persistent region. In the meanwhile, read-in-write operations are allowed again, as shown in S4. The difference is that these read operations should read data from both the persistent region and log block to collect the latest data.

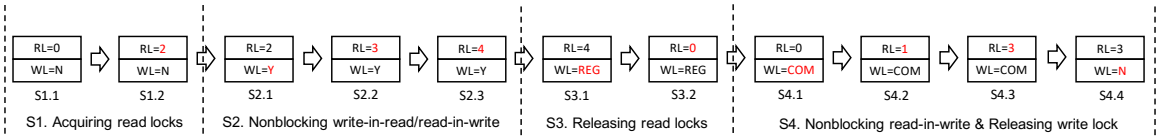
In conclusion, our proposed concurrent access mechanism are coupled with the design of adaptive logging framework, and can flexibly support three types of concurrent accesses: read-in-read, read-in-write and write-in-read operations. Therefore, the concurrent throughput for mixed read-/write workloads will be remarkably improved compared with existing systems that are based on STM [1], [2] or exclusive locks [3], [4].

G. Multiple-Region Transactions

Lewat can support data consistency for multiple-region transactions using the notion of a transaction block. A transaction block can hold multiple *(RID, RAddr)* pairs to record the



(a) Slot region locking



(b) Normal region locking

Fig. 9. Examples of read-write locker modifications.

information of related regions. It maintains *TID* (transaction ID), *RNum* (number of related regions) and *Commit* (commit mark for current transaction) as transaction metadata. When *mtx_start* interface is called, Lewat will allocate and initiate a transaction block and return the transaction ID (TID) to the user. Any subsequent *stx_start* calls will automatically add an (RID, RAddr) pair to the transaction block and update the *RNum* field. The *Commit* field will be marked as `True` only when *mtx_end* interface is called, representing that an integral multiple-region transaction should be completed. The *mtx_end* will implicitly call *stx_end* for each associated persistent region. Therefore, a non-failure return of *mtx_end* indicates that the frontend procedure for all persistent regions is completed. Backend procedure which is out of the critical path will be coordinated by background threads. The transaction block will be recycled when the backend execution for all involved persistent regions is accomplished.

To support correct concurrent access for multiple-region transactions, for simplicity, we currently adopt the conservative two-phase locking protocol [51] which prevents deadlocks. Notice that when a multiple-region transaction detects that a persistent region has been locked by a single-region transaction, it will execute a "register" operation and wait until all the read/write locks are released. On the other side, a single-region transaction can only lock the persistent region when there is no write lock of a multiple-region transaction. A more elegant and efficient concurrent access mechanism for multiple-region transactions is left as our future work.

V. Wear-Leveling

Due to the limited write endurance of emerging non-volatile memory technologies, wear-leveling techniques are critical for PM-based systems to prolong the lifetime of PM device [14], [18], [52], [53]. Numerous wear-leveling techniques have been proposed in the past decade. They can be classified into three categories: write reduction, wear-aware allocation and wear-aware update. Write reduction reduces the number of writes to PM with additional caching layers [15], [54]–[56]. Wear-aware allocation balances the allocations for different PM pages or finer-grained PM cells [29], [57]–[59]. Wear-aware update evens the writes to different PM addresses [52], [60]–[63]. Inspired by these researches, we propose a suite of system-coupled wear-leveling techniques in all three aspects.

Wear-Aware Allocation. Consider an insert-heavy workload that frequently writes newly-allocated persistent regions and quickly recycles the region space. For such a workload, a few hot pages/cells would likely be repeatedly allocated and written. To overcome this problem, we adopt a round-robin allocation method for all allocation units: slots (in a slotted page), pages (in a chunk) and chunks (in a group). Lewat will check the DRAM-resided round-robin marks for pages/chunks before actual allocations occur. For both slots and pages in the two zones, allocation happens in a clockwise style, thus massive allocations can be balanced across different slots and pages. In particular, for small-grained allocation, SAI uses a queue-based management strategy for slotted page allocation (as shown in Figure 1), and hence the small-sized allocation skewness can be further decreased. Similarly, the allocations for replica blocks, log blocks and transaction blocks also conform to the round-robin method. In this way, allocations for PM cells can be even out in Lewat, thus decreasing the wear-out risk caused by allocation.

Wear-Aware Update. Practical workloads can be write-skewed for hot data. For instance, a typical Facebook key-value workload generates 90% updates on less than 10% of the keys [64]. To address this challenge, we develop a two-pronged, user-transparent, wear-aware update mechanism in Lewat. First, the COUL technique designed for major-update workload is inherently wear-aware. COUL allows new writes to persist in a different persistent region. Notice that the allocation for the new persistent region is also wear-aware by complying with the round-robin method, thus skewed writes can be effectively balanced across different PM pages/slots. However, skewed writes in other (non major-update) workloads may still deteriorate the endurance of PM. To figure out this issue, we set a leveling counter (LC) for each persistent region in DRAM to collect the write counts on active regions. When LC exceeds a specified threshold, COUL technique will be triggered compulsively on the next update transaction. Afterwards, LC will be reset for the next round. Assisted with the leveling counter, skewed writes in any types of workloads can be even out due to the judicious usage of the COUL technique. We set the value of LC to 100 in our current Lewat implementation.

Write Reduction. Another issue that may be detrimental to PM is the frequent fixed-address metadata update. For

example, each time a new region is created or an existing region is recycled, the *R_NUM* (i.e., valid region number) in Header area, which is part of the system metadata, should be updated correspondingly. A simple idea is to periodically migrate the metadata fields to different locations. However, it will complicate the management mechanism of our persistent memory system and causes risk of metadata inconsistency. We address this challenge by reducing the fixed-address writes to PM. That is, similar to some existing works [20], [32], [59], [65], we decouple the Lewat metadata (i.e. system metadata and allocator metadata) into critical part (e.g., start offset of slot/page zone, region metadata) and reconstructable part (e.g., allocator bitmaps). Only the critical part needs to be really persisted into PM for every write to guarantee crash consistency; the reconstructable part can be cached in DRAM. Updates to the reconstructable part are directly written to the DRAM-resided copies, which can be persisted back to PM periodically. Therefore, the metadata writes to PM can be remarkably reduced. Such a design will not do harm to data consistency because all the reconstructable can be correctly recovered from the states of the persistent regions.

VI. Crash Recovery

System crash may happen unexpectedly, leading to data inconsistency in a persistent memory system. We carefully devise a recovery algorithm for both single-region transactions and multiple-region transactions to restore the system to a consistent state after system restarts.

When Lewat reboots, it will first check the *MAGIC* field of Header area, which indicates whether the system has exited normally or not. If the system is detected as abnormally-exited, Lewat deals with multiple-region transactions first and then single-region transactions. For multiple-region transactions, Lewat will scan all the transaction blocks. It will continue the execution for each committed transaction and roll back the data of all the involved regions for uncommitted transactions. The order of firstly coping with multiple-region transactions is significant to guarantee data consistency. The reason is that all the data of involved persistent regions in an uncommitted multiple-region transaction should be rolled back even if some involved regions have successfully completed the frontend procedure of single-region write transaction. On the contrary, a single-region transaction does not need to undo the updates once the frontend procedure is completed. The concrete redo and undo procedure for each single persistent region in a multiple-region transaction is similar to the four cases of single-region transactions described below.

For single-region transaction recovery, Lewat will traverse all the persistent regions and for each persistent region, the *LRole* field will be read to determine if inconsistent state exists. After the traversal, all regions marked with inconsistent state with corresponding addresses will be pushed into a queue and a recovery procedure will be conducted for each of them in a parallel scheme using multi-threading. Concretely, there are four recovery cases in Lewat.

(1) The *LRole* field is `SLOT`. This case means that the whole procedure of slot region update is not completed. Lewat will first check the *LRole* field of its replica block. If the state is `END`, then data in the slot region is integral and it will be copied to the replica block to complete the backend procedure; otherwise the write transaction is incomplete, and hence data should be copied from the replica block to the slot region.

(2) The *LRole* field is `FIRST`. It means partial writes to a newly allocated persistent region. Lewat will reset all attributes in the region to be invalid (e.g., memset with 0s).

(3) The *LRole* field is `MAJOR`. This case indicates a half-done procedure for major-update workload. Lewat will check the *LRole* field of its corresponding new region to see if the transaction has completed. If the state is `END`, then the update transaction is committed and the unmodified data should be migrated from the old region to the new region. At last, the old region should be recycled. Otherwise, the new region suffers partial writes and it should be recycled as garbage data.

(4) *LRole* flag is `MINOR`. This implies that a minor-update procedure is being proceeded, with either partial status or intact status. Lewat will verify the log's integrity by 1) checking the *Commit* field and 2) computing the checksum of log records and comparing it with the *Checksum* field. If *Commit* field is `True` or the calculated checksum matches, the transaction is intact thus all the updated data in the log will be written to the corresponding locations in the original persistent region. Otherwise the log block will be dropped.

During the recovery procedure described above, Lewat will also check and update the system metadata (i.e., the Header area) and allocator metadata based on existing valid regions. In addition, it will rebuild all the indexing data structures such as the mapping table and slot allocation index in DRAM when traversing these valid persistent regions.

TABLE III
Non-Volatile Memory Platform Configuration

Processor and Cache	
CPU	Intel Xeon(R) 6240M CPU@2.60GHz
Core Number	32
Private L1 Cache	32KB, 8-way, LRU
Private L2 Cache	1MB,16-way,LRU
Shared L3 Cache	24MB,11-way,LRU
Persistent Memory Attributes	
Capacity	256 GB x 6
Memory Type	AppDirect
Namespace Mode	Filesystem-DAX

VII. Experiments

A. Configurations

We implement Lewat on a Linux server (Ubuntu 18.04.2 LTS, kernel version 4.18.4) with real PM product, Intel Optane DC Persistent Memory (AEP). All experiments are performed on this platform and the system configurations are given in Table III. We create a 128 GB persistent memory device `/dev/pmem7` using Intel's released tools *impctl* and *ndctl*. The entire PM space usage for different zones is initiated as follows: 1 MB for HEAD_ZONE, 100 GB for DATA_ZONE (10 GB for Slot Zone and 90 GB for Page Zone), 1 GB for SR_ZONE, and the remaining 27 GB for LOG_ZONE. The

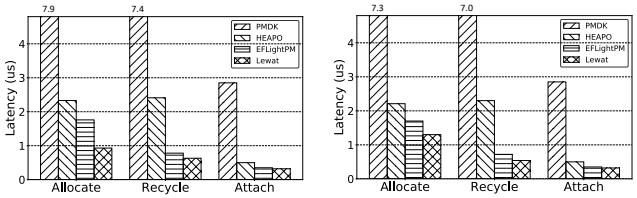
TABLE IV
TYPICAL FEATURES OF MACROBENCHMARKS

YCSB Workload	
YCSB-A	write-heavy, 50% update + 50% read
YCSB-B	read-heavy, 5% update + 95% read
YCSB-C	read-only, 100% read
YCSB-D	read-latest, 5% insert + 95% read
YCSB-F	read-modify-write, 100% (read + update)
Facebook Memcached Workload	
USR	key $\leq$ 24 bytes, value $\leq$ 48 bytes, read = 99.8%, update = 0.2%
APP	key $\leq$ 40 bytes, value $\leq$ 500 bytes, read = 82.5%, update = 5%, delete = 12.5%
ETC	key $\leq$ 52 bytes, value $\leq$ 300 bytes, read = 66.7%, update = 3.3%, delete = 30%
VAR	key $\leq$ 36 bytes, value $\leq$ 100 bytes, read = 18.2%, update = 81.6%, delete = 0.2%
SYS	key $\leq$ 32 bytes, value $\leq$ 800 bytes, read = 66.7%, update = 30%, delete = 3.3%

slot size is configured to be the same as the access granularity of AEP which is 256 bytes [16]. We compare Lewat against three persistent memory systems: PMDK, HEAPO and EFLightPM. The experiments are divided into two parts: micro-test and macro-test. In the micro-test, we compare the metadata and data operation latency for these persistent memory systems. In the macro-test, we compare their transaction throughput and wear-leveling performance using Yahoo! Cloud Server Benchmarks (YCSB [66]) and Facebook Memcached workloads (FB [64]). The features of these two macro benchmarks are shown in Table IV. The *key* size of YCSB workload is fixed as 32 bytes while the *value* size can vary in different tests. Notice that for both YCSB and FB workloads, they are run on the KV-Stores built on persistent memory systems (i.e., PMDK-KV, HEAPO-KV, EFLightPM-KV and Lewat-KV) as a case study. Lewat-KV packs each key-value item as a persistent region. In particular, multiple small items can be stored within the same region. For example, a 256-byte persistent region can contain eight 32-byte key-value items at most. In this way, the request of searching for a specified key is transformed to a search for a specified RID. If the region exists, a further probing in the region for the specified key is required. We also compare Lewat-KV with two state-of-the-art KVs, HiKV [67] and Level-Hash [68]) to verify the efficiency of Lewat-based storage applications. For the compared KV-Stores in our experiments, 72 GB key-value items are first loaded into them as a warming phase, before real workloads are executed. The update size on *value* for each single transaction is randomized between 32 bytes and the full data length. In each of these experiments, ten million operations are executed and the results are averaged over 5 runs. For fair comparison, we also made necessary modifications in other systems. For example, to guarantee persistence ordering, we replace the *clflush+mfence* instructions with more efficient *clwb+sfence* instructions.

B. Comparison of Metadata Access

Figure 10 shows the average latency of a million metadata operations including allocation, recycling and attaching. Concretely, Figure 10(a) and Figure 10(b) give the results for small-sized data (64-byte) and large-sized data (2048-byte), respectively. We can draw three main observations from the results. First, Lewat outperforms other systems in



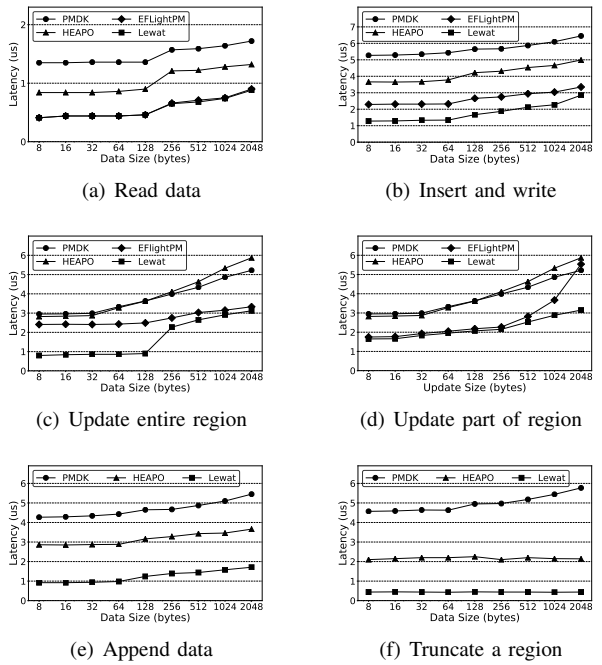
(a) 64-byte (b) 2048-byte

Fig. 10. Latency of metadata access for different data granularities.

all three metadata operations for both small-sized data and large-sized data. It avoids frequent context switches that exist in PMDK and reduces inefficient memory splitting/merging operations that exist in HEAPO and EFLightPM. Second, Lewat has 28.5% lower allocation latency for small-sized data than large-sized data while the recycling performance is comparable for the two granularities. The reason is that small-sized allocation in Lewat can take advantage of the DRAM-resided slot allocation index to locate the best-fitted slots, with merely O(1) searching complexity. For large-sized data allocation, although round-robin marks can help to decrease the searching time, the modifications for layered bitmaps and lockmaps are required, and the region initiation is also more time-consuming. Third, the latency of allocation and recycling for other persistent memory systems are slightly higher in serving small-sized data. This is because small-sized data allocation/recycling with buddy system-like algorithms are more prone to fragmentations than large-sized data, which may cause more split and merge operations [12]. In conclusion, when compared with the counterparts, Lewat produces 24.3%-88.4%, 19.5%-91.7%, 8.6%-88.3% less latency for allocation, recycling and attaching, respectively.

C. Comparison of Data Access

Figure 11 compares the data access latency of the four persistent memory systems. For read operations (Figure 11(a)), Lewat and EFLightPM obtain the lowest latency, outperforming PMDK and HEAPO by 1.9x-3.3x and 1.5x-2.1x, respectively. We argue that the relatively higher read latency in PMDK is caused by the kernel/use layer context switch overhead while that in HEAPO is attributed to the tree-based data indexing. HEAPO adopts a two-layer indexing

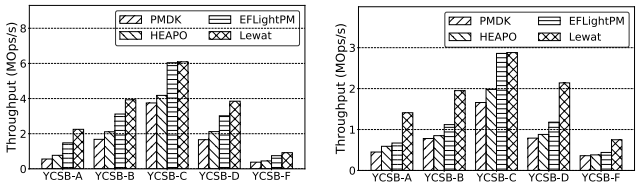


(a) Read data (b) Insert and write

(c) Update entire region (d) Update part of region

(e) Append data (f) Truncate a region

Fig. 11. Latency of data access for different operations.



(a) 64-byte key-value item (b) 2KB key-value item

Fig. 12. YCSB Transaction throughput for different value sizes.

mechanism which utilizes hashing-based structure to organize different clusters and tree-based structure to organize persistent objects in a cluster. In comparison, both Lewat and EFLightPM adopt the pure DRAM-resided hashing-based structure (i.e., mapping table) to index persistent regions. An interesting observation is that for all systems, there is a sharp latency increase between 128-byte data and 256-byte data. This is because the block access granularity of AEP is 256 bytes and persistent region larger than that will cause read amplification.

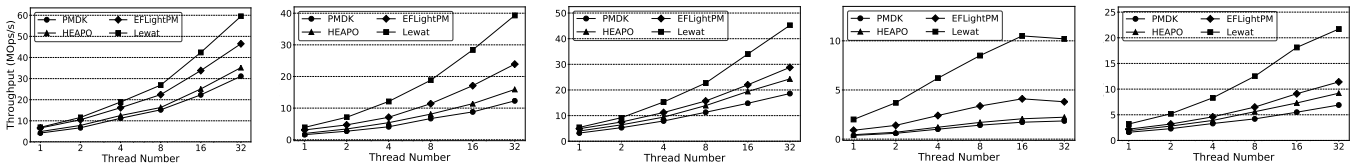
For insert-and-write operation (Figure 11(b)), Lewat gains the lowest access latency, outperforming the other three systems by 2.3x-4.1x (PMDK), 1.8x-2.9x (HEAPO), 1.2x-1.8x (EFLightPM), respectively. Notice that insert-and-write is in fact a combination of allocation and update operations. The higher allocation latency for other systems accounts for the inefficiency in one aspect. The other factor is probably the inflexible logging technique for the first update operation. In Lewat, a region can be recognized as newly-allocated by verifying its *LRole* field, which should be `ALLOC`. For such regions, LOFU technique is automatically utilized and no double write overhead is caused by log records. However, other persistent memory systems such as PMDK still require to write into a log block first, which incurs extra write overhead.

Figure 11(c) and Figure 11(d) show the results for updating an entire region and part of a region, respectively. As the logging scheme of EFLightPM is dependent on manual setting, we set it to COUL for full update operation and remain its default mode, and CERL for partial update operation. From Figure 11(c), we have three main observations. First, for small-sized data (e.g., smaller than 256 bytes), update in Lewat is quite efficient, with 3.0x-3.7x speedup. The reason is that for slot regions, Lewat will automatically utilize REWBL technique, which performs mainly in-place update in the critical path. The other systems have to transform writes to log records in the critical path, and hence logging procedure becomes the bottleneck. Second, for large-sized data, latency of Lewat update is still lower than EFLightPM even though they both employ COUL technique. This follows from the benefit that comes from the faster allocation for new region space assisted with the round-robin markers. Third, for large-sized data, HEAPO has higher update latency than PMDK. This is probably caused by more fences and flushes during the undo logging procedure. In Figure 11(d), the region size is fixed as 2 KB and the update size varies. For partial update, Lewat will adaptively select different logging techniques based on different update sizes. Concretely, for update size smaller than 512 bytes, Lewat will utilize CERL technique, which is the same as EFLightPM. For update size larger than 512 bytes but smaller than 1024 bytes, Lewat will utilize REDO technique. For update larger than 1024 bytes, COUL technique will be chosen by Lewat. EFLightPM does not have an adaptive logging framework and thus can hardly effectively exert the full potential of each single logging technique. In conclusion, Lewat outperforms other systems by up to 1.8x-1.9x when partial update is performed for a persistent region.

Figure 11(e) and Figure 11(f) illustrate the latency of append and truncate operations, respectively. Both the appending size and truncating size is set to be half of the data size in a persistent region. It can be observed that Lewat outperforms two counterparts by 2.1x-4.6x in append and 4.9x-12.9x in trucncate. Different from HEAPO and PMDK, it is not always necessary to allocate new space for appended data in Lewat. Lewat will first check the *Type* field and *AllocNum* field to work out the maximum available storage size. If the free space is sufficient in that region, appended data can be stored and persisted directly adjacent to the tail of the previous data. With strict persistence ordering, an atomic update for *Len* field can ensure data consistency, and hence append can also be conducted as a log-free operation. Truncate in Lewat is as simple as atomically updating the *LRole* field, without any immediate memory recycling procedure. Therefore, Lewat can keep a constant truncate latency whatever the truncate size is. In contrast, PMDK requires memory reallocation and data migration [3], while HEAPO needs to actively recycle freed data space [4]. In conclusion, for both operations, Lewat minimizes the overhead of memory allocation, memory recycling and data migration in the critical path.

D. Comparison of Transaction Throughput

YCSB Workload. Figure 12 shows the transaction throughput of the four persistent memory systems under a variety of

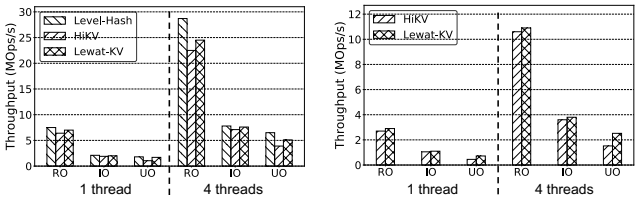


(a) USR (b) APP (c) ETC (d) VAR (e) SYS

Fig. 13. Concurrent transaction throughput of Facebook Memcached workloads.

YCSB workloads for different values sizes. In Figure 12(a), we can observe that for 64-byte value size, Lewat outperforms other systems by 1.27x-2.34x under read-heavy workloads and 1.56x-4.02x under write-heavy workloads. Compared with PMDK and HEAPO, the advantage of Lewat for small-sized data write is that it utilizes REWBL technique and enables lightweight in-place update. As a result, no double write overhead exists in the critical path. The relatively lower throughput of YCSB-F workload for all systems can be attributed to the nature of the read-and-modify workload that causes a write transaction for each single operation. In Figure 12(b), for 2KB value size, we observe that the superiority of Lewat over EFLightPM increases up to 2.11x for write-heavy workloads, compared with 1.56x for the 64-byte value experiment. This is because while EFLightPM statically deals with all write transactions with CERL technique, Lewat adopts an adaptive logging scheme and dynamically picks the best-suited logging technique for each single write transaction. An interesting point of Lewat is that although the throughput of YCSB-D is slightly lower than YCSB-B for small-sized data, the result is reversed for large-sized data. The reason is that for small-sized data, update operations are more efficient with REWBL technique; while for large-sized data, insert operations are more efficient due to LOFU technique, which is automatically activated for the first update on newly-allocated regions. It avoids both the latency overhead and bandwidth consumption of the backend procedure.

Facebook Memcached Workload. The concurrent transaction throughput for Facebook Memacached workloads are illustrated in Figure 13. With a single transaction processing thread, Lewat outperforms its counterparts by 1.05x-1.64x, 1.26x-2.44x, 1.10x-1.69x, 2.12x-5.26x and 1.56x-2.63x for USR, APP, ETC, VAR and SYS, respectively. In particular, we notice that with higher write ratio, Lewat can bring higher performance speedup. With multiple threads, the performance improvement of Lewat can reach up to 1.28x-1.91x, 1.64x-3.20x, 1.60x-2.43x, 2.56x-6.10x and 2.01x-3.29x for five workloads. The reason is that Lewat supports both non-blocking reads and non-blocking writes with its proposed concurrent access mechanism. Therefore, high concurrency is achieved and more transactions can be processed at the same time. An interesting observation is that the throughput cannot scale beyond 32 threads for VAR workload. Since VAR is an update-heavy workload, we believe this is due to the limited write bandwidth of AEP [16].



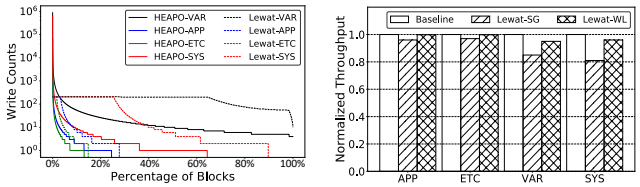
(a) 16-byte key-value item (b) 2KB key-value item

Fig. 14. KV-Store Comparison for Different Item Sizes.

E. Comparison with State-Of-The-Art KV-Stores

To further demonstrate the efficiency of Lewat for building storage applications, we compare the throughput of Lewat-KV with two state-of-the-art KV-Stores: HiKV [67] and Level-Hash [68]. HiKV is a general-purpose KV-Store that supports lookup, range query, insert, update and delete operations. It is based on chain hashing that resolves hash collisions by chaining items as a linked list to the same bucket. Level-Hash is a two-level cuckoo hashing-based index design that can achieve high occupancy, efficient read/write and resizing performance for persistent memory. Both HiKV and Level-Hash implement specific allocators for key-value pairs, but Level-Hash's cacheline-aligned bucket design limits the size of keys and values, and a pointer-based design for large key-value sizes is not directly provided. For fair comparison, we limit the number of used RIDs for Lewat-KV and the load factor (occupancy of allocated persistent memory for KV-Store) is set to 0.6 for all three KV-Stores. Figure 14 shows the throughput under three YCSB workloads: read-only (RO), insert-only (IO) and update-only (UO).

It is observed that Lewat-KV has similar or better throughput compared to HiKV for both small and large key-value sizes under all the three different workloads. For reads, since HiKV adopts a linked list-based hash table design, key probing may traverse a long list based on pointers to find the requested value. As for Lewat, multiple items can be stored in the same region bucket, hence utilizing cache locality and reducing PM read traffic. For updates, HiKV utilizes conventional redo logging, which has higher overhead than Lewat's REWBL technique (for 16-byte items) and COUL technique (for 2KB writes). For small key-value size, the read throughput of Lewat-KV is 7% and 15% lower than Level-Hash for single thread and four threads, respectively. The overhead for key-value reads in Lewat-KV is mainly caused by the indirection layer for the (key, RID) mappings. As for Level-Hash, it can directly probe the target buckets without using an indirection layer. By taking advantage of REWBL technique that masks



(a) Leveling results (b) Performance overhead

Fig. 15. Effect of Wear-Leveling in Lewat.

the background persistence latency, Lewat obtains similar single-thread update update throughput compared with Level-Hash. But with four threads, the throughput of Lewat-KV is 21% lower than Level-Hash. This is because Lewat's REWBL technique also consumes write bandwidth in the backend and write collisions have to wait for the replication to be completed, which degrades the overall performance for multiple threads. In contrast, Level-Hash opportunistically adopts a log-free update scheme that incurs no extra write overhead.

As Lewat is a generic persistent memory system (and not a KV-store), it is not optimized for KV-store like Level-Hash (e.g. optimization for write operations) or HiKV (e.g., concurrency scheme). However, the results in Figure 14 demonstrate that building storage applications atop Lewat can obtain acceptable performance compared to specialized/optimized stores. Furthermore, application programmers do not need to worry about transaction or concurrency mechanisms as Lewat automatically addresses these issues.

F. Wear-Leveling Performance

Figure 15(a) illustrates the effectiveness of Lewat's wear-leveling techniques under four FB workloads. In Figure 15(a), the y-axis shows the write count of each PM block, which is in 256 byte granularity; the x-axis indicates the percentage of PM blocks in decreasing write count order. We choose HEAPO as a representative of existing persistent memory systems that lack software support for wear-leveling. We can observe that HEAPO suffers from skewed writes for all types of workloads. The write count for a few blocks can reach up to a million, while more than 90% of the blocks are hardly modified (fewer than 100 times). In a word, the writes are quite imbalanced and hence the wear-out risk of PM is severe. In contrast, Lewat can effectively distribute the skewed writes across the entire address space. The maximum write count to a single block in Lewat is only 200, which is twice the threshold value of leveling counter. In Figure 15(b), the baseline is Lewat without enabling any wear-leveling techniques. We also port a classic wear-leveling algorithm, start-gap [52], into Lewat (i.e., Lewat-SG) to compare the performance overhead of different wear-leveling designs. We can observe that Lewat-SG reduces the throughput by 4%, 3%, 15%, and 19% for Facebook APP, ETC, VAR and SYS workload, respectively. However, Lewat-WL only introduces up to 5% performance decrease for VAR workload. The reason is that start-gap has to move 16 blocks each time the write threshold is reached and hence the extra write overhead is heavy. In contrast, Lewat-WL only switches the logging technique to balance the new writes when leveling counter is larger than the threshold. Therefore, only a few persistent regions require more background data migration and the extra write overhead is minimized.

VIII. Related Work

Persistent Memory System. Mnemosyne [1], NV-heaps [2], PMDK [3], HEAPO [4], EFLightPM [5] are previous proposals that seek to provide both efficient data management and transaction mechanism. However, these persistent memory systems incur non-negligible bottlenecks, such as context switch overhead, inefficient allocation and recycling, heavy transaction logging, and lack of wear-leveling techniques. LSNVMM [34] employs log-structured organization for persistent memory space and each update is transformed into an appending log. However, the garbage collection overhead and non-sequential reads become the new bottlenecks of its overall system performance. Lewat addresses all these bottlenecks through its decoupled zone allocators, region-centric data management, adaptive logging framework and system-coupled wear-leveling mechanism.

Persistent Memory Allocator. nvm_malloc [31] and Makalu [32] use a three-phase allocation strategy: reserving, initiating and activating. Makalu optimizes the metadata write overhead by considering certain metadata fields as non-critical part that can be recovered after a crash. PAllocator [30] is a fail-safe persistent PM allocator that emphasizes on high concurrency and capacity scalability and exposes three different allocators for different requested sizes. Other allocators such as Walloc [59] and WAFA [29] propose wear-aware allocation technique to extend the lifetime of PM, but they are limited to serve small-size allocations. Our proposed Lewat allocator, however, can provide efficient and scalable service for both small-size and large-size allocations, while supporting wear-leveling and introducing no log overhead.

Data Consistency. Existing persistent memory systems [1]–[4], [21] mainly use redo logging or undo logging to guarantee that the system can be recovered to a consistent state. Some proposals consider reducing the persistence overhead of write transaction in the critical path. For example, SoftWrap [33] and DUDETM [35] keep a shadow data version in DRAM and all updates are first performed on the shadow version. Only on commit, the data is actually copied and persisted into NVM space, assisted by an in-NVM cacheline-optimized redo log. Kamino-Tx [36] maintains an additional copy of each in-transaction data region and perform in-place update directly in the critical path. LP [38] divides a persistent region into multiple LP blocks and each block contains a checksum field. Therefore, no persistence in the critical path is required for new writes. Some proposals tend to reduce the ordering overhead of write transactions. For instance, Kiln [69] introduces a non-volatile cache to remove active flushes to PM. Strand Persistency [13] supports flexible data reordering inside one strand, which may contain multiple data stores. Eager Sync [25] coalesces and reorders writes by deferring commit until locks are released. LOC [70] reduces the commit overhead for writes within a transaction by eliminating the need to perform a persistent commit record write at the end

of a transaction. These works, however, rely on certain cache or memory controller modifications. ATOM [39], ReDU [41] and MorLog [42] remarkably optimizes redo/undo logging in hardware-layer as well. Other proposals, such as Rewind [71], Pronto [72], MOD [37] and RECIPE [73], focus on designing simple and universal guidelines to transform volatile data structures to persistent data structures. Compared to these designs which simply utilize one logging technique for all workloads, Lewat's adaptive logging framework is more comprehensive and lightweight.

Wear-Leveling. Three aspects are covered in this research topic. First, write reduction techniques [15], [54]–[56] reduce the number of writes to PM with additional caching layers such as DRAM and LLC. Skewed writes for hot regions can be directly absorbed in the cache layers. Unfortunately, such cache layers may lead to the risk of data consistency across system crash. Second, wear-aware allocation techniques [29], [57], [58], [74] balance the allocations for different PM pages or finer-grained PM cells. These techniques can work effectively for insert-heavy workload. However, they cannot adequately handle hot regions where frequent updates are concentrated upon. Wear-aware update techniques [52], [60]–[63], [75]–[77] evens out writes across different PM addresses. Therefore, for update-heavy workload, frequent writes can be more uniformly distributed. These wear-leveling techniques are either partial for certain workloads or introduce heavy extra overhead for data consistency. In Lewat, we develop a system-coupled wear-leveling mechanism, which minimizes the extra software overhead for consistency guarantee.

IX. Conclusion

In this paper, we propose Lewat, a lightweight, efficient and wear-aware transactional persistent memory system for fine-grained persistent data management. We decouple the data management for small-sized data and large-sized data to obtain best access performance for each. We propose an efficient adaptive logging framework to minimize the logging overhead for various workloads. Aligned with adaptive logging, concurrency control mechanism and wear-leveling techniques are also designed to enhance the system performance. The experimental results show that Lewat achieves much higher performance than compared systems.

Acknowledgment

This work is supported by National Key Research & Development Program of China (No. 2018YFB1003302), the China Scholarship Council (No. 201906230180) and the National Natural Science Foundation of China (No. 61472241).

References

[1] H. Volos, A. J. Tack, and M. M. Swift, "Mnemosyne: Lightweight persistent memory," in *ACM SIGARCH Computer Architecture News*, vol. 39, no. 1. ACM, 2011, pp. 91–104.

[2] J. Coburn, A. M. Caulfield, A. Akel, L. M. Grupp, R. K. Gupta, R. Jhala, and S. Swanson, "Nv-heaps: making persistent objects fast and safe with next-generation, non-volatile memories," *ACM Sigplan Notices*, vol. 46, no. 3, pp. 105–118, 2011.

[3] "Persistent Memory Programming," http://pmem.io/, 2015.

[4] T. Hwang, J. Jung, and Y. Won, "Heapo: Heap-based persistent object store," *ACM Transactions on Storage (TOS)*, vol. 11, no. 1, p. 3, 2015.

[5] K. Huang, Y. Yan, and L. Huang, "Eflightpm: An efficient and lightweight persistent memory system," in *2019 24th International Conference on Engineering of Complex Computer Systems (ICECCS)*. IEEE, 2019, pp. 154–163.

[6] H.-S. P. Wong, S. Raoux, S. Kim, J. Liang, J. P. Reifenberg, B. Rajendran, M. Asheghi, and K. E. Goodson, "Phase change memory," *Proceedings of the IEEE*, vol. 98, no. 12, pp. 2201–2227, 2010.

[7] K. Wang, J. Alzate, and P. K. Amiri, "Low-power non-volatile spintronic memory: Stt-ram and beyond," *Journal of Physics D: Applied Physics*, vol. 46, no. 7, p. 074003, 2013.

[8] T. Morgan, "Intel shows off 3d xpoint memory performance," 2015.

[9] "Intel optane dc persistent memory: Big memory breakthrough for your biggest data challenges," 2019. [Online]. Available: https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html

[10] S. R. Dulloor, S. Kumar, A. Keshavamurthy, P. Lantz, D. Reddy, R. Sankaran, and J. Jackson, "System software for persistent memory," in *Proceedings of the Ninth European Conference on Computer Systems*. ACM, 2014, p. 15.

[11] R. Kadekodi, S. K. Lee, S. Kashyap, T. Kim, A. Kolli, and V. Chidambaram, "Splitfs: reducing software overhead in file systems for persistent memory," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, 2019, pp. 494–508.

[12] S. M. Rumble, A. Kejriwal, and J. Ousterhout, "Log-structured memory for dram-based storage," in *Proceedings of the 12th {USENIX} Conference on File and Storage Technologies ({FAST} 14)*, 2014, pp. 1–16.

[13] S. Pelley, P. M. Chen, and T. F. Wenisch, "Memory persistency," in *Computer Architecture (ISCA), 2014 ACM/IEEE 41st International Symposium on*. IEEE, 2014, pp. 265–276.

[14] M. K. Qureshi, V. Srinivasan, and J. A. Rivers, "Scalable high performance main memory system using phase-change memory technology," in *ACM SIGARCH Computer Architecture News*, vol. 37, no. 3. ACM, 2009, pp. 24–33.

[15] N. H. Seong, D. H. Woo, and H.-H. S. Lee, "Security refresh: prevent malicious wear-out and increase durability for phase-change memory with dynamically randomized address mapping," *ACM SIGARCH computer architecture news*, vol. 38, no. 3, pp. 383–394, 2010.

[16] J. Yang, J. Kim, M. Hoseinzadeh, J. Izraelevitz, and S. Swanson, "An empirical guide to the behavior and use of scalable persistent memory," in *18th {USENIX} Conference on File and Storage Technologies ({FAST} 20)*, 2020, pp. 169–182.

[17] Y. Chen, Y. Lu, F. Yang, Q. Wang, Y. Wang, and J. Shu, "Flatstore: An efficient log-structured key-value storage engine for persistent memory," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020, pp. 1077–1091.

[18] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger, "Architecting phase change memory as a scalable dram alternative," in *ACM SIGARCH Computer Architecture News*, vol. 37, no. 3. ACM, 2009, pp. 2–13.

[19] K. Bailey, L. Ceze, S. D. Gribble, and H. M. Levy, "Operating system implications of fast, cheap, non-volatile memory." in *HotOS*, vol. 13, 2011, pp. 2–2.

[20] J. Xu and S. Swanson, "Nova: A log-structured file system for hybrid volatile/non-volatile main memories," in *14th USENIX Conference on File and Storage Technologies FAST 16*, 2016, pp. 323–338.

[21] T. Hwang, D. Lee, Y. Noh, and Y. Won, "Designing persistent heap for byte addressable nvram," in *2017 IEEE 6th Non-Volatile Memory Systems and Applications Symposium (NVMSA)*. IEEE, 2017, pp. 1–6.

[22] S. Chu, "Memcachedb: The complete guide," 2008.

[23] J. Zawodny, "Redis: Lightweight key/value store that goes the extra mile," *Linux Magazine*, vol. 79, 2009.

[24] M. A. Olson, K. Bostic, and M. I. Seltzer, "Berkeley db." in *USENIX Annual Technical Conference, FREENIX Track*, 1999, pp. 183–191.

[25] A. Kolli, S. Pelley, A. Saidi, P. M. Chen, and T. F. Wenisch, "High-performance transactions for persistent memories," *ACM SIGPLAN Notices*, vol. 51, no. 4, pp. 399–411, 2016.

[26] A. van Renen, L. Vogel, V. Leis, T. Neumann, and A. Kemper, "Persistent memory i/o primitives," in *Proceedings of the 15th International Workshop on Data Management on New Hardware*, 2019, pp. 1–7.

[27] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz, "Aries: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging," *ACM Transactions on Database Systems (TODS)*, vol. 17, no. 1, pp. 94–162, 1992.

[28] H. Wan, Y. Lu, Y. Xu, and J. Shu, "Empirical study of redo and undo logging in persistent memory," in *2016 5th Non-Volatile Memory Systems and Applications Symposium (NVMSA)*. IEEE, 2016, pp. 1–6.

[29] X. Chen, Z. Qingfeng, Q. Sun, E. H.-M. Sha, S. Gu, C. Yang, and C. J. Xue, "A wear-leveling-aware fine-grained allocator for non-volatile memory," in *2019 56th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2019, pp. 1–6.

[30] I. Oukid, D. Booss, A. Lespinasse, W. Lehner, T. Willhalm, and G. Gomes, "Memory management techniques for large-scale persistent-main-memory systems," *Proceedings of the VLDB Endowment*, vol. 10, no. 11, pp. 1166–1177, 2017.

[31] D. Schwalb, T. Berning, M. Faust, M. Dreseler, and H. Plattner, "nvm malloc: Memory allocation for nvram." *ADMS@ VLDB*, vol. 15, pp. 61–72, 2015.

[32] K. Bhandari, D. R. Chakrabarti, and H.-J. Boehm, "Makalu: Fast recoverable allocation of non-volatile memory," in *ACM SIGPLAN Notices*, vol. 51, no. 10. ACM, 2016, pp. 677–694.

[33] E. R. Giles, K. Doshi, and P. Varman, "Softwrap: A lightweight framework for transactional support of storage class memory," in *2015 31st Symposium on Mass Storage Systems and Technologies (MSST)*. IEEE, 2015, pp. 1–14.

[34] Q. Hu, J. Ren, A. Badam, J. Shu, and T. Moscibroda, "Log-structured non-volatile main memory," in *2017 {USENIX} Annual Technical Conference ({USENIX}{ATC} 17)*, 2017, pp. 703–717.

[35] M. Liu, M. Zhang, K. Chen, X. Qian, Y. Wu, W. Zheng, and J. Ren, "Dudetm: Building durable transactions with decoupling for persistent memory." in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2017, pp. 329–343.

[36] A. Memaripour, A. Badam, A. Phanishayee, Y. Zhou, R. Alagappan, K. Strauss, and S. Swanson, "Atomic in-place updates for non-volatile main memories with kamino-tx." in *EuroSys*, 2017, pp. 499–512.

[37] S. Haria, M. D. Hill, and M. M. Swift, "Mod: Minimally ordered durable datastructures for persistent memory," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020, pp. 775–788.

[38] M. Alshboul, J. Tuck, and Y. Solihin, "Lazy persistency: A high-performing and write-efficient software persistency technique," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2018, pp. 439–451.

[39] A. Joshi, V. Nagarajan, S. Viglas, and M. Cintra, "Atom: Atomic durability in non-volatile memory through hardware logging," in *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2017, pp. 361–372.

[40] S. Shin, S. K. Tirukkovalluri, J. Tuck, and Y. Solihin, "Proteus: A flexible and fast software supported hardware logging approach for nvm," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, 2017, pp. 178–190.

[41] J. Jeong, C. H. Park, J. Huh, and S. Maeng, "Efficient hardware-assisted logging with asynchronous and direct-update for persistent memory," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2018, pp. 520–532.

[42] X. Wei, D. Feng, W. Tong, J. Liu, and L. Ye, "Morlog: Morphable hardware logging for atomic persistence in non-volatile main memory."

[43] J. Arulraj, M. Perron, and A. Pavlo, "Write-behind logging," *Proceedings of the VLDB Endowment*, vol. 10, no. 4, pp. 337–348, 2016.

[44] S. Chen and Q. Jin, "Persistent b+-trees in non-volatile main memory," *Proceedings of the VLDB Endowment*, vol. 8, no. 7, pp. 786–797, 2015.

[45] S. Venkataraman, N. Tolia, P. Ranganathan, R. H. Campbell *et al.*, "Consistent and durable data structures for non-volatile byte-addressable memory." in *FAST*, vol. 11, 2011, pp. 61–75.

[46] J. Liu, S. Chen, and L. Wang, "Lb+ trees: optimizing persistent index performance on 3dxpoint memory," *Proceedings of the VLDB Endowment*, vol. 13, no. 7, pp. 1078–1090, 2020.

[47] S. Zheng, L. Huang, H. Liu, L. Wu, and J. Zha, "Hmvfs: A hybrid memory versioning file system," in *Mass Storage Systems and Technologies (MSST), 2016 32nd Symposium on*. IEEE, 2016, pp. 1–14.

[48] Z. Shao, N. Chang, and N. Dutt, "Ptl: Pcm translation layer," in *2012 IEEE Computer Society Annual Symposium on VLSI*. IEEE, 2012, pp. 380–385.

[49] C. Mitchell, Y. Geng, and J. Li, "Using one-sided {RDMA} reads to build a fast, cpu-efficient key-value store," in *2013 {USENIX} Annual Technical Conference ({USENIX}{ATC} 13)*, 2013, pp. 103–114.

[50] K. Bhandari, D. R. Chakrabarti, and H.-J. Boehm, "Implications of cpu caching on byte-addressable non-volatile memory programming," *Hewlett-Packard, Tech. Rep. HPL-2012-236*, 2012.

[51] P. A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency control and recovery in database systems*. Addison-wesley New York, 1987, vol. 370.

[52] M. K. Qureshi, J. Karidis, M. Franceschini, V. Srinivasan, L. Lastras, and B. Abali, "Enhancing lifetime and security of pcm-based main memory with start-gap wear leveling," in *Proceedings of the 42nd annual IEEE/ACM international symposium on microarchitecture*. ACM, 2009, pp. 14–23.

[53] C. Xu, D. Niu, N. Muralimanohar, R. Balasubramonian, T. Zhang, S. Yu, and Y. Xie, "Overcoming the challenges of crossbar resistive memory architectures," in *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2015, pp. 476–488.

[54] P. Zhou, B. Zhao, J. Yang, and Y. Zhang, "A durable and energy efficient main memory using phase change memory technology," in *ACM SIGARCH computer architecture news*, vol. 37, no. 3. ACM, 2009, pp. 14–23.

[55] G. Dhiman, R. Ayoub, and T. Rosing, "Pdram: A hybrid pram and dram main memory system," in *2009 46th ACM/IEEE Design Automation Conference*. IEEE, 2009, pp. 664–669.

[56] L. E. Ramos, E. Gorbatov, and R. Bianchini, "Page placement in hybrid memory systems," in *Proceedings of the international conference on Supercomputing*. ACM, 2011, pp. 85–95.

[57] H. Aghaei Khouzani, Y. Xue, C. Yang, and A. Pandurangi, "Prolonging pcm lifetime through energy-efficient, segment-aware, and wear-resistant page allocation," in *Proceedings of the 2014 international symposium on Low power electronics and design*. ACM, 2014, pp. 327–330.

[58] H. A. Khouzani, F. S. Hosseini, and C. Yang, "Segment and conflict aware page allocation and migration in dram-pcm hybrid main memory," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 36, no. 9, pp. 1458–1470, 2016.

[59] S. Yu, N. Xiao, M. Deng, F. Liu, and W. Chen, "Redesign the memory allocator for non-volatile main memory," *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, vol. 13, no. 3, pp. 1–26, 2017.

[60] K. Huang, Y. Mei, and L. Huang, "Quail: Using nvm write monitor to enable transparent wear-leveling," *Journal of Systems Architecture*, vol. 102, p. 101658, 2020.

[61] V. Gogte, W. Wang, S. Diestelhorst, A. Kolli, P. M. Chen, S. Narayanasamy, and T. F. Wenisch, "Software wear management for persistent memories," in *17th {USENIX} Conference on File and Storage Technologies ({FAST} 19)*, 2019, pp. 45–63.

[62] K. Zeng, Y. Lu, H. Wan, and J. Shu, "Efficient storage management for aged file systems on persistent memory," in *Proceedings of the Conference on Design, Automation & Test in Europe*. European Design and Automation Association, 2017, pp. 1773–1778.

[63] C.-H. Chen, P.-C. Hsiu, T.-W. Kuo, C.-L. Yang, and C.-Y. M. Wang, "Age-based pcm wear leveling with nearly zero search cost," in *DAC Design Automation Conference 2012*. IEEE, 2012, pp. 453–458.

[64] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny, "Workload analysis of a large-scale key-value store," in *ACM SIGMETRICS Performance Evaluation Review*, vol. 40, no. 1. ACM, 2012, pp. 53–64.

[65] I. Oukid, J. Lasperas, A. Nica, T. Willhalm, and W. Lehner, "Fptree: A hybrid scm-dram persistent and concurrent b-tree for storage class memory," in *Proceedings of the 2016 International Conference on Management of Data*. ACM, 2016, pp. 371–386.

[66] M. Barata, J. Bernardino, and P. Furtado, "Ycsb and tpc-h: Big data and decision support benchmarks," in *Big Data (BigData Congress), 2014 IEEE International Congress on*. IEEE, 2014, pp. 800–801.

[67] F. Xia, D. Jiang, J. Xiong, and N. Sun, "Hikv: A hybrid index key-value store for dram-nvm memory systems," in *2017 {USENIX} Annual Technical Conference ({USENIX}{ATC} 17)*, 2017, pp. 349–362.

[68] P. Zuo, Y. Hua, and J. Wu, "Write-optimized and high-performance hashing index scheme for persistent memory," in *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, 2018, pp. 461–476.

[69] J. Zhao, S. Li, D. H. Yoon, Y. Xie, and N. P. Jouppi, "Kiln: Closing the performance gap between systems with and without persistence support," in *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, 2013, pp. 421–432.

[70] Y. Lu, J. Shu, L. Sun, and O. Mutlu, "Loose-ordering consistency for persistent memory," in *Computer Design (ICCD), 2014 32nd IEEE International Conference on*. IEEE, 2014, pp. 216–223.

[71] A. Chatzistergiou, M. Cintra, and S. D. Viglas, "Rewind: Recovery write-ahead system for in-memory non-volatile data-structures," *Proceedings of the VLDB Endowment*, vol. 8, no. 5, pp. 497–508, 2015.

[72] A. Memaripour, J. Izraelevitz, and S. Swanson, "Pronto: Easy and fast persistence for volatile data structures," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020, pp. 789–806.
[73] S. K. Lee, J. Mohan, S. Kashyap, T. Kim, and V. Chidambaram, "Recipe: converting concurrent dram indexes to persistent-memory indexes," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. ACM, 2019, pp. 462–477.
[74] S. Yu, N. Xiao, M. Deng, Y. Xing, F. Liu, Z. Cai, and W. Chen, "Walloc: An efficient wear-aware allocator for non-volatile main memory," in *2015 IEEE 34th International Performance Computing and Communications Conference (IPCCC)*. IEEE, 2015, pp. 1–8.
[75] X. Chen, E. H.-M. Sha, X. Wang, C. Yang, W. Jiang, and Q. Zhuge, "Contour: A process variation aware wear-leveling mechanism for inodes of persistent memory file systems," *IEEE Transactions on Computers*, 2020.
[76] Q. Liu and P. Varman, "Ouroboros wear leveling for nvram using hierarchical block migration," *ACM Transactions on Storage (TOS)*, vol. 13, no. 4, pp. 1–31, 2017.
[77] D. Liu, T. Wang, Y. Wang, Z. Shao, Q. Zhuge, and E. Sha, "Curling-pcm: Application-specific wear leveling for phase change memory based embedded systems," in *2013 18th Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE, 2013, pp. 279–284.

APPENDIX A
COMPARISON FOR PM ALLOCATORS

Table V compares Lewat allocator against several state-of-the-art PM allocators: nvm_alloc [31], Makalu [32], PAllocator [30], WAlloc [59] and WAFA [29]. Notice that in Table V, small-allocation indicates allocation size that is smaller than 4KB, big-allocation means allocation size larger than 4KB.

nvm_malloc uses a three-step allocation strategy, namely reserving memory space, initializing it and finally activating it. nvm_malloc uses a segregated-fit algorithm for blocks smaller than 2 KB, and a best-fit algorithm for larger blocks. Makalu's allocation scheme is similar to nvm_malloc, but differs in that it enforces the persistence of only a minimal set of metadata, and reconstructs the remaining during recovery. It relies on offline garbage collection to relax constraints of metadata persistence, thus supporting fast small-block allocations. These two allocators, however, mainly focus on the scalability of small-size allocations, but neglect that of large-size allocations. By contrast, PAllocator uses three different allocation strategies (i.e., SmallPAllocator, BigPAllocator, HugePAllocator) for different allocation sizes. Concretely, similar to nvm_malloc and Makalu, SmallPAllocator uses a segregated-fit allocation algorithm. BigPAllocator is a FPTree-based [65] allocator that uses best-fit algorithm. HugePAllocator creates one segment back by a persistent file per allocation, and deletes that segment on deallocation. For concurrency, PAllocator maintains one SmallAllocator and one BigPAllocator object per core, while maintaining only one HugePAllocator, as huge allocations are bound by the file system performance.

WAlloc [59] and WAFA [29] are two PM allocators optimized for wear leveling. WAlloc minimizes metadata writes through separating volatile metadata and non-volatile metadata. It allocates NVM space with less-allocated-first-out policy. By redirecting data writes around, it also prevents some specific memory areas from being hot spot under changing workloads. In WAFA, the PM pages are divided into basic units and contiguous units can satisfy the allocation requests with corresponding size in rotation, hence the space efficiency is better than the slab-based techniques [1], [3], [30]–[32], [59] with frequent splits and merges. The basic units are alternatively utilized to reduce the wear of pages caused by small writes. However, they are only designed to serve small-grained data allocations (e.g., several bytes to a few hundreds of bytes), which limit their use for general-purpose applications. To ensure the allocation and deallocation atomicity, all these mentioned PM allocators should utilize logs, which add extra write and persistence overhead for PM allocators.

Lewat's slot zone allocator absorbs the design of WAFA to enable high performance for small-size data allocation, but does not introduce any logging overhead for allocator consistency because it does not need to maintain the linked list structure of slotted pages in PM. All the fields in the metadata can be constructed from the states of data slots (i.e., whether they are valid persistent regions). For large-sized data allocations, Lewat's page zone allocator uses a layered bitmap. Compared with the best-fit algorithm for large free blocks [3], [31], [32], Lewat allocator is more time-efficient because an atomic 8-byte bitmap update can allocate data space ranging from 4KB (one page) to 1GB (64 groups). The bitmap can be stored in DRAM space and can be recovered by page zone scanning, which distinguishes free pages from valid persistent regions. Therefore, the page zone allocation is also log-free. To support high scalability, we maintain per-thread slot allocation index for small-size allocations while partitioning chunks to different threads for large-sized allocations. In conclusion, Lewat allocator is time-efficient, scalable for all allocation sizes. It is also wear-aware and simple to use, which can act as a lightweight and comprehensive allocator for all PM-based storage applications.

We conduct experiments to verify the performance benefit of Lewat allocator over other PM allocators. The results are shown in Figure 16. It can be observed that Lewat allocator achieves the best throughput (per thread) for all different allocation sizes. Concretely, for small-allocations (figure 16(a) and 16(b)), Lewat allocator outperforms others by 1.06x-7.79x. For big-allocations (figure 16(c), 16(d) and 16(e)), Lewat allocator reaches 2.35x-2.71x throughput over PAllocator, the best existing PM allocator for large-sized allocations.

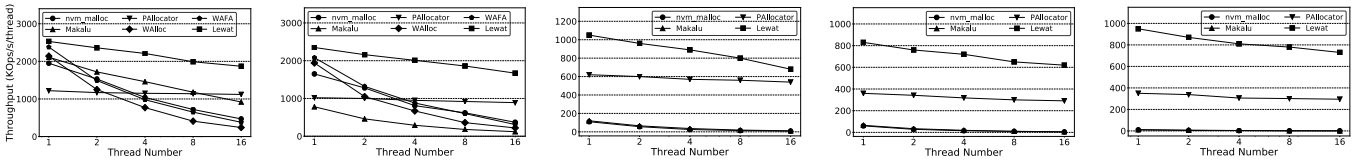
APPENDIX B
COMPARISON FOR TRANSACTION MECHANISMS

Table VI compares Lewat adaptive logging with several state-of-the-art software transaction mechanisms, including SoftWrap [33], DudeTM [35], LSNVMM [34], Kamino-Tx [36] and LP [38]. Notice that in table VI, small-insert/small-update workload indicates write size that is smaller than 512B, large-insert/large-update workload means write size larger than 512B.

Both SoftWrap and DUDETM utilize DRAM to cache a persistent region as a shadow version for write batching. During writes, cacheline-optimized redo logs are generated and persisted into PM in an append-only method. Meanwhile, the updates are performed in the DRAM-resided shadow version. When a transaction should be committed, all the updates in the shadow version will be persisted back to the original persistent region in PM asynchronously in the background.

TABLE V
COMPARISON OF PERSISTENT MEMORY ALLOCATORS

	nvm_malloc [31]	Makalu [32]	PAllocator [30]	WAlloc [59]	WAFA [29]	Lewat
small-allocation-efficient	✓	✓	✓	✓	✓	✓
big-allocation-efficient	×	×	✓	/	/	✓
scalable	×	×	×	×	✓	✓
log-free	×	×	×	×	×	✓
wear-aware	×	×	×	✓	✓	✓
simple-for-use	×	✓	×	✓	✓	✓



(a) 128 B (b) 1 KB (c) 8 KB (d) 64 KB (e) 256 KB

Fig. 16. Throughput Comparison for Different Allocation Sizes.

TABLE VI
COMPARISON OF TRANSACTION MECHANISMS

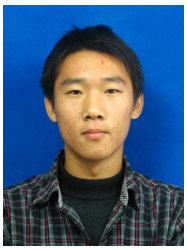
	SoftWrap [33]	DudeTM [35]	LSNVMM [34]	Kamino-Tx [36]	LP [38]	Lewat
small-insert-efficient	✓	✓	✓	✓	✓	✓
large-insert-efficient	×	×	✓	✓	×	✓
small-update-efficient	✓	✓	✓	✓	/	✓
large-update-efficient	✓	✓	✓	×	/	✓
read-efficient	✓	✓	×	✓	×	✓
space-efficient	×	×	×	×	✓	✓
scalable	×	✓	✓	×	×	✓
wear-aware	×	×	✓	×	×	✓

The differences between SoftWrap and DUDETM are that 1) SoftWrap uses region-level mapping while DUDETM uses page-level mapping and 2) SoftWrap directly copies data from shadow memory to PM on commit while DUDETM copies data from the per-thread redo log. Therefore, in both mapping and data persistence procedure, DUDETM can achieve better performance by eliminating unnecessary writes for unmodified data. However, for each single transaction, the persistent region may be forced to be copied to DRAM space regardless of the write size. This can be inefficient for large data update.

Kamino-Tx [36] maintains an additional copy of each data region and any modifications to a persistent region will be first performed as in-place update. When a transaction is committed, the new writes should be copied to the replica regions out of the critical path. However, for large-update workloads, this technique can be inefficient due to heavy background writes, which can consume a lot of PM write bandwidth. LSNVMM adopts log-structured organization for the entire PM space and treats each write as an append-only log. In this way, any modifications to the persistent regions are transformed to sequential writes for logs, which can achieve high write performance, while automatically balancing skewed writes evenly across PM addresses. The tradeoff, however, is read efficiency (due to scattered data storage and pointer-based indirections) and space consumption (obsolete data will exist before garbage collection). Similar to DUDETM, LSNVMM also uses per-thread log for better scalability. LP (i.e., Lazy Persistence) uses checksum to eliminate the persistence overhead caused by *clwb+mfence* in the critical path. Concretely, it divides a persistent region into multiple smaller LP blocks, each with a separate checksum which indicates the consistency of that single LP block. However, LP is not a log-based transaction mechanism and all writes are directly performed on the original persistent regions. Therefore, during unexpected system crash, partial updates will corrupt certain LP blocks. The checksum can only tell if a LP block is inconsistent but cannot assist to recover to a consistent state. Therefore, LP mechanism is limited in use for certain programs that can regenerate consistent state by re-execution or workloads that contain no update operations.

Different from previous transaction mechanisms that use a single technique for all types of workloads, Lewat's adaptive logging framework utilizes different logging techniques and dynamically selects the appropriate scheme based on the features of the workload. For insert-write workloads, Lewat automatically utilizes LOFU technique, which introduces no write amplification. For small-update workload, Lewat automatically utilizes REWBL technique for slot regions and CERL technique for other larger regions, both of which incur eliminates persistence overhead off the critical path. For large-update workload, Lewat will automatically utilizes COUL or REDO technique to minimize the write bandwidth consumption. Therefore, as shown in our experimental study, Lewat's adaptive logging offers first-class performance for

all different types of workloads. We also believe that Lewat can naturally work well for changing workloads without any source code modifications.



Kaixin Huang is currently a Ph.D. student at Shanghai Jiao Tong University, China. He received his bachelor degree from University of Electronic Science and Technology of China, in 2016. His research interests include non-volatile memory management and distributed systems.

Sumin Li is currently working toward the Ph.D. degree in the department of computer science and engineering at the Shanghai Jiao Tong University (SJTU). Her research interests lie in the area of architecture-driven software development, system software, high performance computing, in-memory computing. and big data analysis.

Linpeng Huang received his M.S. and Ph.D. degrees in computer science from Shanghai Jiao Tong University in 1989 and 1992, respectively. He is a professor of computer science in the department of computer science and engineering, Shanghai Jiao Tong University. His research interests include distributed systems and service oriented computing.

Kian-Lee Tan is a Professor of Computer Science at the School of Computing, National University of Singapore (NUS). He received his Ph.D. in computer science in 1994 from NUS. His current research interests include query processing and optimization, database performance, data science, and databases on modern hardware. Kian-Lee was a member of the VLDB Endowment Board (2012-2017) and PVLDB Advisory Committee (2014-2017). Kian-Lee is a member of ACM and a senior member of the IEEE.

Hong Mei is a professor of computer science at Shanghai Jiao Tong University and Peking University. He obtained his Ph.D. degree from Shanghai Jiao Tong University, in 1992. His main research interests range over software engineering and system software. Hong Mei is a fellow of the IEEE and TWAS, a member of Chinese Academy of Sciences and a foreign member of the Academia Europaea.