On the Performance Intricacies of Persistent Memory Aware Storage Engines

Zhiwen Chen *, Wenkui Che *, Daokun Hu, Xin He, Jianhua Sun, and Hao Chen †

Abstract—As key components of DBMSs, various storage engines and index structures have been proposed based on incorrect assumptions before PMem hardware is publicly available. Recent studies reveal that there is a significant performance gap in evaluating index structures on real PMem platforms as compared to DRAM-based emulators. However, a comprehensive evaluation for those PMem-aware database storage engines on real PMem hardware is still missing. Meanwhile, dynamic memory management is more important on PMem systems because PMem is slower than DRAM and unfriendly to random small-writes, and ensuring crash-consistency for the metadata of PMem allocators introduces extra overhead. Therefore, it is essential to understand the performance intricacies of PMem-aware database storage engines from the perspective of PMem allocators. This paper presents a systematic evaluation of three PMem-aware database storage engines using representative workloads and a unified benchmarking framework that is integrated with four PMem allocators. Besides the commonly used metrics, the impact of different hardware configurations (such as NUMA and eADR) on performance is also considered. Through in-depth analysis, we reveal caveats and pitfalls on using or designing PMem-aware storage engines and important insights that can serve as guidelines for future development of PMem allocators and other related components.

Index Terms—Persistent memory, non-volatile memory, NUMA, storage engine, memory allocator.

1 Introduction

NOWADAYS, data-intensive applications are becoming increasingly common, and the scale and characteristics of workloads in production systems are changing significantly. In terms of scale, the data to be processed by popular applications is growing exponentially, and in terms of characteristics, workloads are shifting from read-intensive to write-intensive [1] as well as featuring high concurrency. These changes disrupt the delicate balance that systems strive to maintain between quality of service, service-ability, and costs. Software systems urgently need new memory technologies to escape the current dilemma. Fortunately, persistent memory (PMem) featured with large capacity, byte-addressability, persistence, as well as DRAM-scale latency was available and offers opportunities to solve the problems that many systems have been facing. To date, a plethora of PMem technologies based on different physical mediums have been proposed [2], [3], [4], [5], [6], [7]. 3D-XPoint is one of the well-known PMem technologies, and Intel Optane DC Persistent Memory Modules (hereinafter referred to as DCPMM) based on this technology is the first-to-market PMem product.

Database management system (DBMS) can benefit from the new features of PMem to achieve higher performance as well as instant recovery, as it no longer needs to write data to slow non-volatile devices, such as SSDs or HDDs, to retain data after system crashes. However, one cannot directly port traditional DBMS to a PMem platform, as the optimizations it adopts may become inefficient or even invalid when applied to PMem systems. It is also stressed that DCPMM should not simply be treated or used as a *slower, persistent DRAM* [8], and blindly applying traditional disk or DRAM based approaches would not reap its full benefits [9]. As a result, researchers have started to rethink the designs of PMem-aware DBMS.

Prior to the availability of PMem hardware, several storage engines [10], [11], [12] and a variety of index structures [13], [14], [15], [16], [17], [18], [19], [20], [21], [22] have been proposed, which are the key components of a PMem-aware DBMS. As the performance of PMem hardware has been extensively profiled [8], [23], [24], [25], [26], more important details about it are revealed, and some findings are inconsistent with previous assumptions. For instance, the authors of [27] identify that PMem bandwidth is scarcer than expected and has a significant impact on performance, and authors of [8] find that the end-to-end write latency is often lower than read latency since write pending queues (WPQs) are protected by the asynchronous DRAM refresh (ADR) domain, which are both contrary to prior assumptions. The latest research [27] shows that existing persistent indexes based on incorrect or invalid assumptions can only achieve suboptimal performance. The follow-up studies [1], [9], [22], [28] promptly adapt their methodologies in designing index structures that could truly exploit the performance dividends afforded by PMem. However, it remains unexplored how those database storage engines proposed in emulation environments would perform and differ on real PMem systems. Furthermore, the importance

- *Z. Chen, W. Che, D. Hu, J. Sun, and H. Chen are with the College of Computer Science and Electronic Engineering, Hunan University, Chang Sha 410082, China. Email: {zhiwenchen, chewenkui, daokunhu, xinhe, jhsun, haochen}@hnu.edu.cn*
- *Z. Chen is also with the School of Computer Science & School of Cyberspace Science, Xiangtan University, Xiang Tan 411105, China.*
- **The first two authors contributed equally to this work.*
- †*H. Chen is the corresponding author.*

Manuscript received 12, April, 2022; revised 23 January, 2023.

of dynamic memory management is highlighted by the trend of the increasing proportion of write operations in production workloads [29], [30]. The authors of [31] share an insight that memory allocation presents significant compute cost at the warehouse scale, and its optimization can yield considerable cost savings. And a recent study [32] shows that dynamic memory allocation consumes nearly 7% of all cycles in Google datacenters. This ratio would be higher in PMem systems, because PMem is slower than DRAM, and ensuring crash consistency for allocator related metadata on PMem introduces additional overhead. Therefore, to circumvent the performance issues and design pitfalls for future PMem allocators and PMem-aware storage engines, it is of paramount importance to understand the performance intricacies of PMem-aware database storage engines from the perspective of PMem allocator on a real PMem platform.

To this end, we conduct a comprehensive and in-depth study of PMem-aware database storage engines on real PMem systems. We first port `Nstore` [10], a database benchmarking framework based on a PMem hardware emulator, to real PMem systems to make it run correctly and efficiently. Then, we integrate four state-of-the-art PMem allocators covering different design spaces, including Makalu [33], nvm_malloc [34], NVAlloc [35] and PMDK [36], into `Nstore` with necessary extensions. Next, we evaluate the performance of three PMem-aware database storage engines based on different PMem allocators from both macro and micro perspectives. In addition to the commonly used metrics such as throughput, thread scalability, tail latency, execution overhead, LLC misses, and recovery overhead, we also consider the impact of NUMA architecture and eADR (extended ADR) supported by new hardware. The source code of the benchmarking framework is publicly available on GitHub[1]. Our experimental results obtained from running representative workloads reveal important insights that can be adopted as guidelines for the future development of PMem-aware storage engines, PMem allocators, and other correlated components. We summarize our findings below, and more details will be discussed in Section 5. ① Beware of the performance restrictions imposed by random PMem writes. ② Memory allocators consume about 36% CPU cycles in PMem-aware database storage engines. ③ Tailoring PMem allocator for achieving maximum performance. ④ NUMA-awareness is desirable to achieve higher read performance. ⑤ eADR is not a panacea for better performance. ⑥ No size fits all. ⑦ Co-design of PMem-based indexes with real storage systems.

The rest of this paper is organized as follows. Section 2 provides background on PMem and PMem allocators. In Section 3, we review the design and implementation of PMem-aware storage engines, and present our extensions to port them to a real PMem platform. The evaluation results are discussed in Section 4. We summarize observations in Section 5. Related work and conclusion are presented in Section 6 and Section 7.

1. https://github.com/HNUSystemsLab/pm-engine

2 Background

2.1 Persistent Memory

Architecture. DCPMM was introduced with the second generation Intel Xeon Scalable Processors (codenamed as Cascade Lake) [37], [38]. The Cascade Lake processors support multiple sockets (2/4/8) with each containing one or two processor dies that comprise separate NUMA nodes [8], [39]. The DIMMs for both DCPMM and DRAM are controlled by the processor's integrated memory controllers (iMCs). The iMCs communicate with DCPMM in cacheline (typically 64 bytes) granularity, while the 3D-XPoint physical medium has coarser internal access granularity (256 bytes). Therefore, when persisting data to PMem, the on-DIMM controller translates smaller requests into larger 256-byte ones, causing write amplification as small stores become read-modify-write operations. The on-DIMM controller has a small write-combining buffer (referred to as XPBuffer [8]). Since not the whole memory hierarchy in the system is persistent, so systems supporting PMem today must have an Asynchronous DRAM Refresh (ADR) mechanism. Several components of the iMCs, including the XPBuffer and write pending queues (WPQs), are protected by ADR. The feature of ADR ensures that CPU stores that reach the ADR domain will survive a power failure [40]. Unfortunately, ADR protects the iMCs but leave out the processor caches [23], [41]. Programmers must inject cacheline flush (such as `clflush`, `clflushopt`, and `clwb`) and memory fence instructions (such as `mfence` and `sfence`) explicitly to ensure data persistence at the right time and in the desired order, which makes applications error-prone [42], [43], [44], [45] and may incur significant performance degradation. To address this problem, the third generation Intel Xeon Scalable Processors (codenamed Copper Lake) have an extended ADR (eADR) that incorporates CPU caches into the protection of the ADR domain. With eADR, no flushing is necessary (memory fence is still needed), which greatly simplifies PMem programming [41].

Performance. DCPMM has lower bandwidth and higher read and write latency compared to DRAM. As reported in previous work [24], the random read latency of DCPMM is ~400 ns, which is ~3.3x of that of DRAM (~120 ns). Moreover, DCPMM exhibits asymmetric read/write latency with write significantly slower than read at the device level. However, according to another study [8], the end-to-end write latency *as seen by the application* is often lower than read latency, because writes can return once data reaches the ADR domain at the memory controller. But reads have a high probability of accessing the physical medium if the data is not cached in read pending queues (RPQs). Random writes have ~4x higher latency than sequential writes. The maximum sequential read and write bandwidth of a single DCPMM is about 6.6GB/s and 2.3GB/s [8], [39], [46] respectively, which is 6.3% and 3.1% of the bandwidth of a DDR4 DRAM. When there are six interleaved DCPMMs, the platform can provide sequential read and write bandwidths of up to 40GB/s and 13GB/s, respectively [21]. The asymmetric bandwidth is more prominent for random reads and writes [9], [27], [39].

2.2 Persistent Memory Allocators

Memory allocator is one of the essential components for building high-performance applications. On DRAM-based systems, the memory allocators are usually well tuned to achieve low response latency, high scalability, and efficient memory utilization. However, for PMem allocators, in addition to the need to maintain salient features of DRAM allocators to provide efficient memory management [35], they also need to ensure crash consistency so that the memory heap can be restored to the latest consistent state after unexpected system errors and power failures. In this part, we focus on reviewing the design space of four representative PMem allocators in achieving high-performance memory management and ensuring crash consistency, including Makalu [33], the allocator in Intel Persistent Memory Development Kit (PMDK) [36], nvm_malloc [34], and NVAlloc [35].

Makalu [33] is a PMem allocator that adopts a crash consistency mechanism based on garbage collection. Similar to the memory allocator proposed in [47], Makalu's heap is structured as a *Big-Bag-of-Pages*. For ease of management, it subdivides the heap into 4KB blocks and only maintains necessary metadata at the block level and in separate headers. The heap metadata of Makalu is divided into core metadata and auxiliary metadata. Core metadata is irrecoverable once corrupted and requires strong consistency guarantees. In contrast, the auxiliary metadata can be reconstructed from consistent core metadata when corrupted or lost, and Makalu adopts a garbage collection mechanism to relax the persistence constraints on this kind of metadata. This approach enables Makalu to achieve a low persistence overhead per allocation. This is because for each memory request, Makalu only needs to persist the core metadata in a way that always guarantees its consistency, rather than to ensure consistency of all corresponding metadata. Makalu's management of the persistent memory heap consists of two distinct phases, i.e., the online and offline phrase. After initialization, Makalu starts the online processing phase, during which it handles memory allocation requests from the application and records log entries to ensure the crash consistency when updating the core metadata. For high performance memory management, Makalu uses different allocation strategies for small ($\leq$ 400 bytes), medium (between 400 bytes and 2KB), and large memory (> 2KB) objects. Specifically, small-sized objects are allocated from the thread-local freelist. Medium-sized objects are allocated from the appropriate global freelist that is guarded by a lock, and large objects are serviced directly from the chunk (a contiguous sequence of blocks) freelist. After a system crash or power-failure, Makalu enters the offline phrase. It first launches log replay to restore the core metadata to a consistent state, from which the auxiliary metadata can then be reconstructed. After the user data of the application is restored consistently, it performs garbage collection to detect and fix persistent memory leaks.

PMDK is a collection of persistent memory programming libraries developed by Intel [36], which is widely used to implement PMem systems [9], [48], [49]. We focus on the transactional persistent allocator of PMDK (provided by the `libpmemobj` library). The heap of PMDK is reconstructed lazily and in stages when recovering from system crashes. The entire heap is divided into equally-sized zones, and metadata is placed at the beginning of each zone for better cache locality. Each zone is further divided into memory blocks, called chunks, of different sizes to reduce memory fragmentation. Whenever there is an allocation request and the runtime state indicates that there is no enough available space to satisfy it, the metadata of the zone is processed, and the corresponding runtime state is initialized. This approach helps to minimize the startup time of the pool and amortize the cost of reconstructing the heap state across many individual allocation requests. Memory allocation in PMDK is performed in two steps. The first step is called reservation. It reserves the state that is needed to perform the operation. Taking a memory allocation as an example, at this step, PMDK retrieves a free memory block, marks it as reserved, and initializes the object's contents. The second step is the act of exercising the reservation, which is called publication. PMDK adopts different allocation schemes for objects with different sizes. For the allocation of objects larger than 256 KB, PMDK adopts a best-fit algorithm, and the information of these large-sized objects is managed through an AVL-tree. While the allocation of small-sized objects is served by a segregated-fit algorithm with 35 size classes, and the information of small-sized objects is managed by freelists, which are placed in DRAM and implemented as vectors of pointers to persistent memory blocks. When performing memory operations, a redo logging mechanism is adopted to ensure the heap can be recovered from system failures correctly.

nvm_malloc [34] is a strongly consistent PMem allocator based on a WAL crash consistency model [35]. It provides `malloc`/`free`-like interfaces to manage persistent memory. The general design of nvm_malloc is similar to the jemalloc [50] allocator. The heap consists of 4 MB chunks, each of which can be logically divided into multiple 4 KB blocks. Allocations are split into three major size classes: small (under 2 KB), large (between 2 KB and 2 MB), and huge (larger than 2 MB). Huge and large allocations directly use one or more contiguous chunks and blocks, respectively. Small allocations are subdivided into subclasses that are multiples of aligned 64-byte requests. To easily locate allocated regions when recovering from system crashes and to avoid permanent memory leaks, nvm_malloc uses a two-step process (reserve and activate), which is similar to PMDK. In the reserve step, the actual allocation is performed by finding or creating a free space large enough for the request. In the activate step, it takes the address of the allocated region and retrieves the according header section. For large and huge objects, the link pointers are written into the respective header on PMem and the state is modified to indicate the pending activation. For a small object, nvm_malloc needs to write the link pointer and the corresponding bitmap, followed by a memory fence, state change, and flush. It maintains volatile copies of metadata called `VHeaders` in DRAM. VHeaders maintain a pointer that points to their persistent equivalent and are used to track regions in trees or hashmaps in DRAM. In this way, all read-only operations can be executed on the volatile copies instead of accessing the PMem directly.

NVAlloc [35] is another persistent variant of the jemalloc allocator. The difference is that it focuses on solving the performance issues caused by repeatedly flushing the same CPU cache line and random small-sized writes to PMem when updating and maintaining heap metadata, and reducing memory fragmentation caused by static slab segregation. It consists of a small allocator and a large allocator. The small allocator is responsible for serving allocation requests smaller than 16KB. It inherits the design of arena and thread-local cache (tcache) from traditional volatile memory allocators to alleviate synchronization overhead. In addition, the small allocator implements a slab structure, which has a persistent header and a volatile header (called vslab). The persistent header stores the metadata (for example, a bitmap indicating the state of each block of the current slab) that is necessary for recovery, while the volatile header is used to speed up the process of searching for free blocks. NVAlloc uses an interleaved mapping from data blocks to their corresponding heap metadata and interleaved layout of linked list in tcaches to avoid flushing the same CPU cache line frequently in a short period of time. The large allocator is responsible for allocating memory from extents for requests ranging from 16KB to 2MB. The extents are memory chunks allocated from the persistent heap. For efficient management of extents, NVAlloc maintains three virtual extent header (VEH) lists in DRAM, including (1) an activated list stores the VEHs of allocated extents, (2) a reclaimed list stores the VEHs of freed extents with physical persistent memory being mapped to virtual addresses, and (3) a retained list stores the VEHs of free extents that only have virtual addresses allocated and their physical memory has been unmapped in the process address space. VEH is a copy of the metadata of the persistent extent. During allocation and release, the corresponding VEH is updated, and its core metadata is appended to a persistent bookkeeping log to ensure crash consistency. For objects larger than 2MB, NVAlloc calls `mmap()` to allocate space. Upon a failure recovery, NVAlloc rebuilds vslabs and VEHs based on the core metadata in the extent headers and slab headers, respectively.

3 Benchmark Design and Implementation

In this section, we first briefly introduce the implementation of three PMem-aware database engines proposed in `Nstore` [10] that is evaluated using an emulation environment. Next, we discuss how to extend `Nstore` in order to adapt these storage engines to work in a real PMem system.

3.1 PMem-Aware Database Storage Engines

`Nstore` [10] is a lightweight framework developed to evaluate different PMem storage architecture designs for OLTP workloads. It supports a pluggable storage engine backend. Therefore, we can fairly compare the performance characteristics of different storage engines, memory allocators, and crash consistency mechanisms under a unified hardware and software environment. It implements three PMem-aware database storage engines: the In-Place (InP) engine, the Log-Structure Merged (LSM) engine, and the Copy-on-Write (CoW) engine, which are based on the in-place, log-structured, and copy-on-write update strategies, respectively. All the three engines support four primitive operations (i.e., *select*, *insert*, *update*, and *delete*). To prevent data corruption in case of power loss or system failures, the engines can call a recovery function to undo all operations in a transaction to guarantee consistency. The data stored in `Nstore` is organized as a series of records with each consisting of fixed-sized and variable-sized fields. The fixed-sized field stores the pointers to different types of data (e.g., int, double, and string), while the variable-sized field stores the actual data. To leverage persistent memory, `Nstore` implements a memory allocator that keeps metadata of every block in a header located just before the user data region. Each memory request will be processed by this memory allocator. These three engines mainly differ in the way they index records and the strategy they adapt to ensure consistency, which are discussed in the following.

InP Engine. The InP engine gets its name from the way it updates the fields of an existing tuple, i.e., the system writes the new value directly on top of the original value. It stores tuples and non-inlined fields using fixed-size and variable-length blocks, respectively. To index tuples, the InP engine maintains a B+tree, which resides completely in PMem and uses the same interface as the engine to access PMem. Note that for the InP engine implemented in NStore, all operations that alter the B+tree's internal structure are atomic, which means that the B+tree does not incur additional consistency overhead. The InP engine employs write-ahead log (WAL) to ensure crash consistency, and stores the log entries as a linked list. A new log entry will be appended to the list using an atomic write. Specifically, the InP engine writes a log entry into PMem before the transaction performs any modifications. Each entry contains the transaction identifier, the table modified, the tuple identifier, and the address of the tuple being modified (or the address of the field being updated). When the logging is completed, a pair of `clflush` and `sfence` instructions is issued to ensure that the content of the log is written to PMem. After all the transaction's changes are safely persisted, InP truncates the log file to clear up the expired log entries, then issues an `sfence` instruction to make sure the truncate operation is done before the next transaction starts.

The four operations supported by the InP engine are shown in Algorithms 1-4, which include not only the common parts of the three engines but also statements specific to one engine (distinguished by **if-else** conditions). All the operations will go through a decoding phase first, in which the engine can obtain the information that is necessary to store the data correctly. Then, the index key of the B+tree structure is calculated. In terms of the supported operations, when the *update* operation is done, the non-inlined fields of the corresponding tuple will be modified. The *insert* and *delete* operation will alter the index structure. The *delete* operation will clear the tuple exactly as can be seen in Algorithm 2 line 17. The InP engine will delete some other data when committing a transaction. The delete operation only removes the index node.

LSM Engine. The LSM engine is a write-optimized design that uses an LSM tree for data management. The LSM tree consists of multiple levels, and each level contains a sorted run of data. The topmost level is called MemTable, which is used to store the entries that record the changes performed on tuples by transactions. When the MemTable

Algorithm 1 Nstore's basic database operation: insert.

```
 1: function INSERT(statement)
 2:     Get tuple and table_id from statement
 3:     index = get_index(table_id)
 4:     key = hash_function(tuple)
 5:     if index.find(key) is true then ▷ duplicate key detected.
 6:         clear tuple
 7:         return failure
 8:     else
 9:         if engine type is LSM or InP. then
10:             allocate space for log_entry
11:             log_entry.write(tuple)
12:             log_list.add(log_entry)
13:             persist(log_list)
14:         allocate space for Index node
15:         index.insert(node) ▷ LSM only insert MemTable's index
16:         persist(index)
17:     return success
```

Algorithm 2 Nstore's basic database operation: delete.

```
 1: function DELETE(statement)
 2:     Get tuple and table_id from statement
 3:     index = get_index(table_id)
 4:     key = hash_function(tuple)
 5:     if index.find(key) is true then
 6:         if engine type is LSM or InP. then
 7:             allocate space for log_entry
 8:             log_entry.write(tuple)
 9:             log_list.add(log_entry)
10:             persist(log_list)
11:             index.delete(key)
12:         else                                  ▷ CoW engine's case
13:             delete_record = index.get(key)
14:             index.delete(key)
15:             persist(index)
16:             clear delete_tuple          ▷ clear data immediately
17:         clear tuple
18:     else                                          ▷ key not found.
19:         return failure
20:     return success
```

Algorithm 3 Nstore's basic database operation: update.

```
 1: function UPDATE(statement)
 2:     Get tuple, update_field and table_id from statement
 3:     index = get_index(table_id)
 4:     key = hash_function(tuple)
 5:     if index.find(key) is true then ▷ LSM searches 2 indices.
 6:         if engine type is LSM or InP. then
 7:             allocate space for log_entry
 8:             log_entry.write(tuple)
 9:             log_list.add(log_entry)
10:             persist(log_list)
11:             tuple.modify(update_field)
12:             persist(tuple)
13:         else                                  ▷ CoW engine's case
14:             allocate space for copy_tuple
15:             copy_tuple = tuple
16:             copy_tuple.modify(update_field)
17:             persist(copy_tuple)
18:             alter index structure
19:             persist(index)
20:             delete tuple
21:     else                                          ▷ key not found.
22:         return failure
23:     return success
```

Algorithm 4 Nstore's basic operation: select.

```
 1: function SELECT(statement)
 2:     Get string_key, select_field and table_id from statement
 3:     key = hash_function(string_key)
 4:     index = get_index(table_id)
 5:     if index.find(key) is true then
 6:         value = tuple.get_value(select_field)
 7:         return value
 8:     return failure
```

is full, the LSM engine marks it as immutable and creates a new MemTable. For convenience, we refer to an immutable MemTable as an SSTable. SSTables are physically stored in the same way on PMem as the MemTable. The only difference is that the engine does not propagate writes to the SSTables. The LSM engine performs a compaction process to generate a larger SSTable (logically) by merging the index structures of a set of SSTables. In addition, the LSM engine maintains a bloom filter for each SSTable to filter out negative lookups. The two approaches above help to mitigate read amplification. The LSM engine adopts WAL to ensure crash consistency for the MemTable, i.e., rebuilding the MemTable and undo the effects of uncommitted transactions from the MemTable during recovery.

The four operations in LSM engine are shown in Algorithms 1-4. Note that the LSM engine uses distinct B+trees for the MemTable and SSTables to track records. To *insert* a tuple (lines 9-16 of Algorithm 1), the LSM engine first scans the MemTable and all SSTables to ensure no duplicate keys exist. Then, it flushes the tuple to PMem and records the tuple pointer in a log entry. Next, it persists the log entry and marks the tuple as persisted. Finally, it adds an entry in the MemTable's B+tree. When a transaction *delete*s a tuple, the LSM engine deletes the node corresponding to the tuple from the index, and then marks the tuple as a tombstone (line 11 of Algorithm 2). The space of tombstones is reclaimed during the compaction of SSTables. The *update* operation follows the same procedure as the *delete* operation.

CoW Engine. Similar to the InP engine, the CoW engine has separate pools for fixed-sized and variable-length data. It maintains the persistent state of each slot in both pools. This engine always maintains two directories, including (1) a current directory that points to the most recent versions of the tuples and only contains the effects of committed transactions, and (2) a dirty directory that points to the versions of tuples being modified by uncommitted transactions. There is no need for the CoW engine to maintain a log because the original value is never changed until the new one is created. The index structure of the CoW engine is a sophisticated B+tree, which provides transaction guarantee inside the index structure. The CoW engine spawns a background thread that periodically commits all modifications. To make sure that the background thread has done its job, the main thread will check the lock state at the start and end of transactions.

As shown in Algorithms 1-4, the *insert* and *select* operation are straightforward to implement for the CoW engine as it does not need any logging and the new element does not need a shadow copy to maintain a consistent state. In contrast, the *update* operation is relatively complicated, as shown in lines 14-19 of Algorithm 3. To update a tuple, the CoW engine first makes a copy of the tuple to be updated, then applies changes to this copy and stores the pointer of the modified tuple in the dirty directory of the CoW B+tree. To amortize the cost of persisting the current directory, the CoW engine batches transactions, during which it persists the changes performed by uncommitted transactions and the contents of the dirty directory sequentially, and then updates the master record using an atomic durable write to point to that directory. The master record is the root node of the current directory of the index structure, which is always guaranteed to be in a consistent state. Note that the CoW engine uses a background thread to reclaim the memory of old records, the index operation of the front thread is only in charge of generating new page and redirecting pointers.

3.2 Extensions to Nstore

`Nstore` was only evaluated in an emulation environment, which may lead to inaccurate and even misleading results. In addition, some design choices made in `Nstore` may neglect the importance of certain components like memory management. In this regard, the main goal in this work is to re-evaluate the performance of database storage engines on real PMem hardware. However, adapting existing emulation-based systems like `Nstore` to real PMem hardware is not trivial. In the rest of this section, we discuss several technical challenges relevant to PMem management and implementation and our solutions in detail.

Memory Management. Although `Nstore` uses thread-local parameters in the engines, the entire PMem file remains locked during each allocation and deallocation of PMem memory, which is actually unnecessary in most cases. This is because mapped PMem memory can be divided into several sub-sections that can be used to serve requests from different threads independently. There exist some PMem allocators with built-in concurrent control, such as Makalu [33] and nvm_malloc [34]. Due to the performance issue of the default allocator in `Nstore`, we redesign its memory management interface to allow to plugin different PMem allocators such as Makalu, nvm_malloc, PMDK[2] and NVAlloc[3]. In this way, we can evaluate the performance impact of these PMem allocators under a variety of workloads flexibly.

We make the following modifications. First, we remove the original locking mechanism in the memory subsystem of `Nstore` and leave the concurrent control to the PMem allocators. Second, Makalu uses a private thread activation function to make some allocator-related initialization after a thread is spawned. To reserve this function, we add two interfaces *pmem_thread* and *pmem_join* for this purpose if Makalu is used as the PMem allocator. Otherwise, threads are managed by the std::thread library directly. Third, in order to persist data from cache, we allocate 8 more bytes for every object, assisting `Nstore` in determining the size flushed to PMem by the `clflush` instruction. These extra allocated bytes are necessary and will be used to guarantee crash consistency. Although this approach incurs some space overhead, it is tolerable and has a marginal impact on performance because data objects are often padded to align with a cacheline (or multiple cachelines). In fact, existing PMem designs commonly introduce additional metadata to ensure crash consistency, such as the bitmaps and version counters maintained in B+trees and hash tables.

Memory Related Issues. As PMem is byte addressable and typically mapped into a fixed size of memory address space [4], there are some problems that may not be easily exposed in an emulation-based system, such as false usage of persistent pointer and metadata corruption. We take a typical bug as an example to demonstrate this issue.

nvm_malloc and PMDK use a header per block to maintain metadata, this strategy works in the case of DRAM because the physical DRAM address space is discontinuous and any illegal memory accesses will trigger a segmentation fault immediately. However, this is not the case with PMem, because the PMem memory is fully mapped to the user space and the operating system does not immediately block any accesses to and modification of the PMem, which may lead to further data overwriting. Taking nvm_malloc as an example, it retains some state information in the header to determine whether the PMem space is reachable and usable. The header is 64 bytes long and generated when a new block is created. Suppose users needs a 4035-byte object, the memory cost of each block should be 4035 plus 64 bytes, which means 4099 bytes in total need to be aligned. In practice, two blocks are used in nvm_malloc and thus 8192 bytes are allocated for this request. However, in the implementation of nvm_malloc, the aligning algorithm mistakenly allocates only `one` block for such a request. In fact, the blocks are consumed in reverse order per chunk in this implementation, which means the last 64 bytes of an object may be written to the header used by a previous request, as shown in Figure 1. This metadata corruption problem does not cause the entire system to halt immediately because the next block is still in the mapped address space of PMem and may not be used immediately. But this can be problematic in further use, being more difficult to track and locate. In our implementation, we fix this problem by adding 64 more bytes to ensure that two blocks are allocated for the same user request.

Avoiding Slow File System Operations. In the original implementation of `Nstore`, the performance of the LSM engine is suboptimal because it stores SSTables in separate files and accesses them through slow file operations such as `fseek` and `fwrite`. To solve this, we store each SSTable as an array that is managed by PMem allocators. They are maintained in the same heap file as the MemTable. We chose

2. We use PMDK-1.11.
3. We use its log-based version.

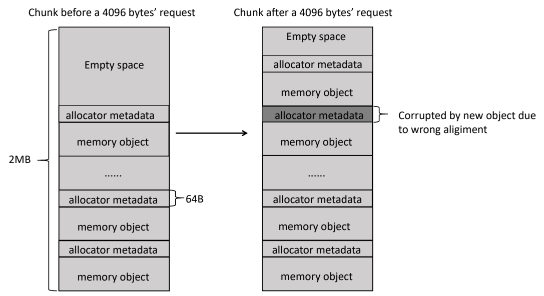

Chunk before a 4096 bytes' request
Chunk after a 4096 bytes' request
Empty space
allocator metadata
memory object
......
64B
2MB
Corrupted by new object due to wrong aligiment

Fig. 1. Memory overflow in nvm_malloc.

TABLE 1
YCSB workloads. The suffix numbers indicate the ratios of operations. The prefix characters `I`, `R`, `U` and `D` means *insert*, *search*, *update*, and *delete* respectively. For example, `R100` means a workload with 100% search operation.

Workloads							
a	b	c	d	e	f	g	h
R100	R90 I10	R50 I50	R10 I90	R90 U10	R50 U50	R10 U90	D100

array rather than other data structures for three reasons. (1) It can reduce query time. By using arrays, the target element can be quickly retrieved based on the starting memory address, the length of each element, and the offset. (2) The size of elements in SSTables is fixed because each of them is a <key, pointer> pair of a tuple in the storage pool. This feature allows us to easily transform an SSTable file into an SSTable array. (3) We do not need to redesign the index of SSTables (i.e., we use the original B+tree in a SSTable as the index), since the array structure is also indexed by integer values. The trade-off is that when merge happens, extra `clflush` and `sfence` instructions are needed. Because new records inserted in the MemTable should be flushed to the corresponding SSTable during merging. In this paper, the array buffer is empirically set to be 80 MB per thread, which is big enough for our benchmarks because we observe that the SSTables are less than 40 MB in all cases. This is a configurable parameter and can be adjusted according to the workload.

4 Evaluation

In this section, we evaluate the performance of the three storage engines with two widely used benchmarks, YCSB [51] and TPC-C [52]. We use four general-purpose PMem allocators, including Makalu, nvm_malloc, PMDK, and NVAlloc, to replace the default allocator in `Nstore` [10] to dissect the impact of PMem allocators. For simplicity, in the following we use the abbreviation of the engine name with suffixes -P, -M, -N and -A to represent different combinations of engines with allocators such as PMDK, Makalu, nvm_malloc, and NVAlloc. For example, CoW-M denotes a CoW engine that uses Makalu as the memory allocator. When necessary, we conduct extra experiments to measure microscopic metrics, including the amount of (1) read/write bytes per operation and (2) LLC misses, to provide a more fine-grained and in-depth analysis. Besides the two metrics, i.e., thread scalability and tail latency, we also evaluate the performance on recovery and the performance of storage engines with the next generation PMem hardware architecture (particularly the extended ADR domain).

We do not include the results for other PMem allocators such as PAllocator, Ralloc, and Poseidon for the following reasons. Ralloc [53] is a persistent allocator transformed from a non-volatile lock-free allocator. Similar to Makalu and NVAlloc, it uses post-crash GC to relax the consistency constraints online. The experimental results show that the performance of the two allocators, Ralloc and NVAlloc, is close to each other because they share a lot common designs. Therefore, to avoid too many allocator+engine combinations affecting readability, we omitted the results of Ralloc. PAllocator [54] is a fail-safe PMem allocator focusing on reducing fragmentation, as well as improving concurrency and capacity scalability. Poseidon [55] is a PMem allocator to guarantee heap metadata protection and achieve high performance and scalability. However, we exclude them as they are not either general purpose designs or publicly available. For instance, Poseidon relies on the Memory Protection Keys (MPK), a set of hardware primitives only supported by specific Intel processors, to achieve heap metadata security.

4.1 Environment and Setup

We perform our experiment on a dual-socket Linux server with kernel version of 5.4.0. The server is equipped with two Intel Xeon Gold 5218 CPUs that are clocked at 2.3 GHz and have 16 physical cores (32 threads), 32 KB L1 instruction cache, 32 KB L1 data cache, and 1024 KB L2 cache. The last level cache is 22 MB in size. The system has 192 GB (6 x 32 GB) of DDR4 DRAM and 768 GB (6 x 128 GB) of Optane DCPMM in total. The Optane DCPMMs are configured in `AppDirect` mode. Except the evaluation of NUMA, only one CPU is used and the DCPMMs are all installed with this CPU, and the threads are also bound to it.

4.2 Workload

YCSB. Previous work [10] evaluates the performance of PMem-aware database storage engines with four types of mixed workload that represent different ratios of `read` and `update` operations (i.e., `a, e, f, and g` in Table 1). Besides these workloads, we consider three workloads mixed with read and insert operations (`b, c, and d` in Table 1) to test the insert performance. The workloads contain a single table comprised of tuples with a primary key (8 bytes) and 5 columns of random strings (each is 200 bytes), so each tuple is approximately 1 KB in size. To measure the throughput of read, insert, and update, we preload the engines with 10 million tuples and then execute 10 million operations to perform the measurements. To evaluate the throughput for the delete workload (`d` in Table 1), we first initialize a table with 10 million tuples, then delete them until the table is empty.

TPC-C. The TPC-C benchmark simulates an order-entry environment on a wholesale supplier and is the current industry standard for evaluating the performance of OLTP systems [56]. TPC-C consists of five types of transactions as listed in Table 2. The Delivery transaction executes one

TABLE 2
TPCC transaction details.

Trans. Name	Op. Kind			
	Insert	Remove	Update	Select
Delivery	✗	✓	✓	✓
New Order	✓	✗	✓	✓
Order Status	✗	✗	✗	✓
Payment	✓	✗	✓	✓
Stock Level	✗	✗	✗	✓

TABLE 3
The amount of LLC misses per read operation. Measured using a read-only workload with 16 threads.

	InP	CoW	LSM
Makalu	15.1	21.8	17.6
PMDK	17.5	21.4	18.1
nvm_malloc	18.0	24.9	19.9
NVAlloc	22.5	27.4	22.9

remove, several select and update operations. It simulates a situation where a company selects a client's order, updates its balance, and removes order from order lists. The New Order transaction selects several items from the warehouse, and generate a new order which will be inserted into order tables. The Payment transaction requires the engine to select a customer from the customer table first. Then, it processes the payment of money, which is actually an update operation in payment table. Finally, it inserts this payment movement into the history table. For convenience, we classify a transaction as *write-type* if it contains any insert, delete, or update operations. Delivery, New Order, and Payment are write-type transactions. Order Status and Stock Level only involve query requests, so we call them *read-type* transactions. Their execution time grows with the scale of the table. The difference between Order Status and Stock Level is that after the select operation, the Stock Level transaction performs a filter operation to pick out all items that are below a predefined number. Similar to prior work [10], we configure the workload to contain eight warehouses and each has 100K items, with all transactions evenly assigned to the workers.

4.3 Thread Scalability

4.3.1 YCSB Performance

Figure 2 shows the throughput of the 8 workloads listed in Table 1. Sub-figures in the same row correspond to the results obtained using the same allocator. All results are the average of five runs.

Read-Only Workload. We first analyze the results of read-only workload illustrated in Figure 2(a). The throughput of all engines scales well. The read performance of the LSM engine is inferior to the InP engine. This is because the former is a design optimized for write-intensive workloads, and it needs to access the indexes of all the runs of the LSM tree that contain records associated with the desired tuple in order to reconstruct the tuple. For each read, the CoW engine fetches the master record and then finds the tuple in the current directory, which make it perform the worst with the read-only workload. To illustrate more clearly, we measure the average number of bytes accessed per select operation. From the results shown in Figure 3(b), we can observe that the CoW engine reads more bytes from PMem than LSM and InP. Interestingly, for the read-only workload that does not involve any dynamic memory management, the throughput of InP-A is lower than that of InP-P. We attribute this to the fact that the spatial locality of addresses allocated to user data by different memory allocators in not the same. For instance, NVAlloc adopts interleaved mapping to avoid repeated flushes to the same cache line. However, this approach results in the address space allocated to user data being discontinuous (i.e., poor spatial locality). From Table 3, we can see that InP-A has more LLC misses than InP-P.

Mixed Workload of Insert and Read. The most notable observation from this workload is that Makalu is unfriendly to the two log-based engines. With higher percentage of insert operations, the impact of Makalu on the overall performance of InP and LSM is more noticeable. As shown in the first row of Figure 2(b)-(d), the maximum throughput of InP-M and LSM-M is only 44.9%/58.4% and 24.4%/8.1% of that of CoW-M on balanced and insert-intensive workloads, respectively. We attribute this to the incompatibility of the memory usage behavior of storage engines with the internal design of the PMem allocator. For the log-based engines, the PMem allocator needs to allocate space for the newly created log entries and the key-value pair when inserting a tuple. Log entries and key-value pairs are small objects, and the required memory space is allocated from the allocator's fast path. Existing PMem allocators usually use thread local cache (or thread local freelist for Makalu), a design derived from jemalloc, on their fast paths to speed up small allocations. When the thread-local cache for a particular size is empty, the allocating thread scans the reclaim list or the global block list corresponding to that size to refill it after acquiring the global lock, which is time-consuming. Compared to other allocators, Makalu performs this time-consuming refilling process more frequently. This is because it assigns only a 4 KB block to the thread-local freelist (for a particular size) at initialization, while the size of the thread local cache of NVAlloc, PMDK, and nvm_malloc is 64 MB, 4 MB, and 2 MB, respectively.

With PMDK, we have three observations. (1) The performance gap between the three engines narrows as the percentage of insert operations in the workload increases. (2) The throughput of all engines scales almost linearly with the number of threads and peaks at 31 threads. (3) InP lags behind LSM and CoW in throughput when more write operations are involved in the workload. Since both PMDK and nvm_malloc are based on jemalloc [50] and share some optimizations and designs, as compared to PMDK, the throughput of all engines nvm_malloc scales similarly. The difference is that with nvm_malloc, InP achieves the highest throughput with the balanced workload, and CoW outperforms others with the write-intensive workload.

Mixed Workload of Update and Read. In this scenario, the CoW engine exhibits the worst performance in all configurations. The reason is that when a transaction updates a tuple, the engine first makes a copy and then applies the changes to that copy. On the one hand, this approach

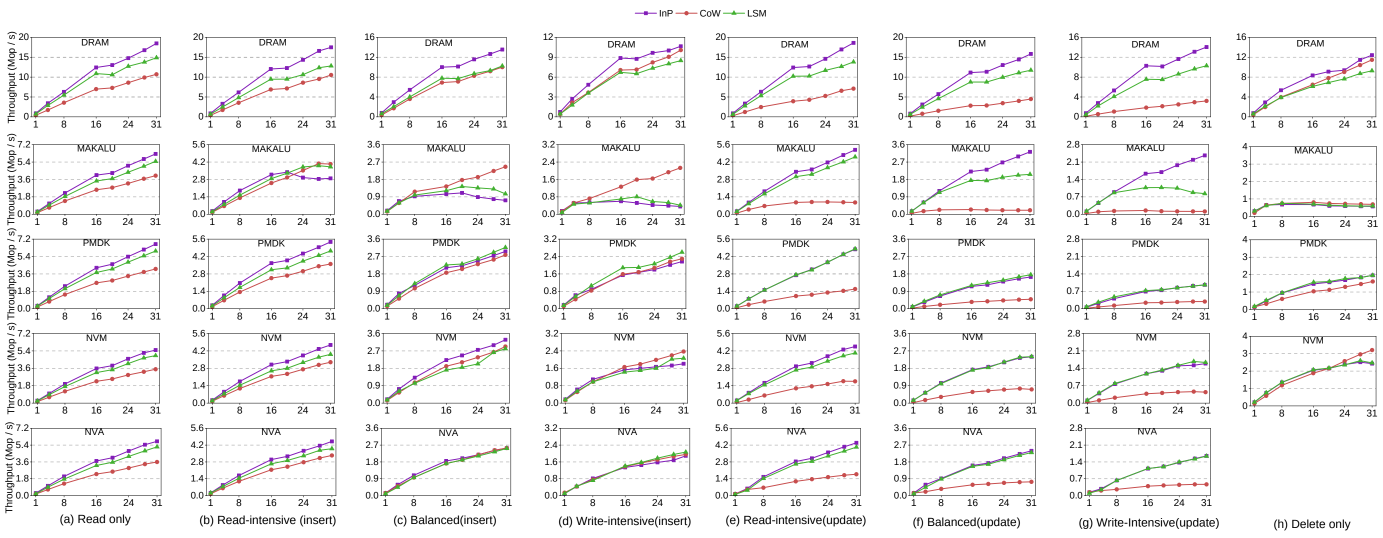


Fig. 2. YCSB throughput. Each column corresponds to a workload in Table 1.

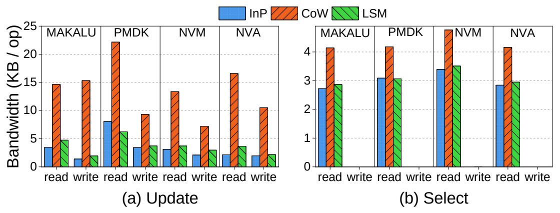


Fig. 3. The average number of bytes of write/read accessing to PMem in 2 typical operations.

incurs more PMem accesses than other engines, i.e., more severe PMem write amplification, as shown in Figure 3(a). On the other hand, the CoW engine amplifies the number of required expensive PMem allocations (to allocate space when creating the copies of tuples). We discuss the overhead of memory management in Section 4.5.

Next, we discuss the performance variation with different allocators. The first row of Figure 2(e)-(g) shows that LSM and CoW based-on Makalu both suffer from thread scalability issues as the update ratio increases. The reason for the former is that update operations may trigger the merge operation of LSM, in which all log entries generated after the last merge operation are cleaned up, and the corresponding space is reclaimed. Makalu supports thread-local deallocation, the freed memory objects will be firstly returned to a thread local freelist, which will then add them to the *global reclaim list* (requires global locking) when the number of free blocks in the local freelist is beyond a threshold. The higher ratio of update operations in the workload would degrade the performance more significantly for LSM based on Makalu. The issue with CoW is caused by lock contention during memory allocation, since the space for the copied data page in CoW when updating a tuple is allocated from the *global freelist* of Makalu that is protected by a lock. The second and third row of Figure 2(e)-(g) show the results when using PMDK and nvm_malloc. We can observe that LSM and InP perform almost identically (in most cases, their curves overlap). However, the LSM and InP engine based on the nvm_malloc achieve a higher throughput than their PMDK-based counterparts, and the higher the percentage of updates in the workload, the more noticeable difference in throughput. This is because PMDK has more complex processing logic and internal data structures than nvm_malloc, which incurs higher overhead for creating log entries for update operations.

Delete Only Workload. As shown in Figure 2(h), when using Makalu, the thread scalability is severely restricted, with throughput peaking at 8 threads. Given that there is no lock contention inside the storage engine, so the thread scalability problem stems from the design of Makalu. Similar to the update workload discussed above, the *global reclaim list* is the main reason for poor thread scalability. The *global reclaim list* resides in DRAM and maintains the list of free blocks to be distributed among thread local freelists [55]. As a result, intensive lock contention is produced on the global reclaim list when numerous objects need to be reclaimed. Worse, Makalu adopts a different method from PMDK and nvm_malloc in reclaiming large objects. It clears the space of medium- or large-sized objects with `memset`, while PMDK and nvm_malloc only need to modify their heap metadata (such as the state of a block in the freelist and bitmaps), which is cheaper than the strategy employed by Makalu. LSM reclaims multiple log entries and pointers during the merge process. Therefore, the thread scalability of LSM is less affected compared to InP in this case. When using PMDK or nvm_malloc, the performance of InP is almost equal to that of LSM. Compared to PMDK, nvm_malloc can provide higher throughput on delete only workload. With nvm_malloc, InP, LSM, and CoW achieve 1.23x, 1.32x, and 1.99x higher throughput than their PMDK-based counterparts respectively. Note that the engines using NVAlloc fail to run the delete-only in a multithreaded scenario. This is caused by read-write conflicts on the global red-black tree (RBTree), which is an internal index structure of NVAlloc and is used to accelerate the log operations during memory allocation/release. Therefore, the sub-figure in the last row of Figure 2(h) is not shown.

Variable-Length Tuples. To evaluate the performance of the engines when processing the workload with variable-length tuples, we construct a new insert-read balanced YCSB workload consisting of a mixture of records with sizes varying between 1 and 256 bytes (denoted as YCSB-var). Figure 4 presents the results. In this case, the engines exhibit similar performance trends as evaluated with YCSB-c, except higher throughput.

Upper-bound Performance. We measure the upper-bound throughput by allocating memory from DRAM. Results are shown in the first row of Figure 2. Similar to the results obtained from emulator-based evaluation, the InP engine performs the best across a wide set of workloads. In the read-only scenario, the performance curves of the three engines on DRAM and PMem exhibit similar trends, but the former has about 3 times higher throughput on average, which is approximately equal to the performance gap between DRAM and PMem. However, for write workloads, the throughput with DRAM is up to 30x higher than that of the PMem counterparts (the case of InP-M when running YCSB-d), which is significantly larger than the write performance gap between DRAM and PMem. We attribute this mainly to the extra overhead introduced by ensuring crash consistency.

4.3.2 TPC-C Performance

Although TPC-C is also an update-intensive workload, its behavior is somewhat different from YCSB-g in two aspects. First, the operations in a TPC-C transaction are executed on tables with different sizes. Second, a transaction consists of multiple write operations, each with variable size (the size of a record ranges from tens to hundreds of bytes). Figure 5 displays the results. We use the number of transactions performed per second (`txn/s`) to represent the throughput. In addition, we also measure the upper-bound throughput of the engines by allocating memory from DRAM.

From Figure 5, we obtain the following observations. First, the CoW engine performs poorly on the TPC-C workload (no matter which allocator is used) for the same reason as in the case of YCSB-g. Second, NVAlloc is able to provide the engines with throughput closer to their DRAM counterparts (about 33%). These two observations are consistent with the update-intensive case of YCSB (see Figure 2(g)). Third, LSM outperforms InP in this case, while their performance is almost identical under YCSB-g. In addition, the throughput of LSM-N drops at 28 threads. This is mainly due to the impact of persist instructions. By removing flush/fence instructions, it can scale to higher throughput with more threads.

Lastly, all engines face thread scalability issues with Makalu to some extent. Concretely, the throughput of InP-M, LSM-M, and CoW-M peaks at 20, 16, and 8 threads, respectively. This is caused by the memory reclamation the engine performed after transactions are committed. During the execution of a write-type transaction (e.g., a Payment or New Order transaction), the engine reclaims the space of expired log entries and dirty fields after all modifications have been persisted to PMem. A single log entry and a dirty field are usually small is size, and Makalu needs to return the reclaimed memory blocks to a thread-local freelist to fulfill future memory allocations. When the thread-local freelist has sufficient free blocks, it will add them to the global reclaim list. Clearly, the performance of InP-M is affected more significantly than that of LSM-M, because they take different strategies to reclaim the space of expired log entries. Specifically, InP clears the expired log entries immediately after all modifications are persisted, while LSM clears them only when a merge operation is triggered, which is less frequent than transaction commit. Therefore, LSM-M suffers less from the global lock contention in Makalu.

4.3.3 Summary and Discussion.

We summarize the results as follows. First, the performance of database storage engines varies with the type of workload, and no specific engine can always be superior in all scenarios. This is inconsistent with the conclusion made in prior study [10] that claims *the PMem-aware InP engine performs the best across a wide set of workloads*.

Second, the write throughput that an engine can achieve based on different allocators varies greatly, even when running the same type of workload. Since a PMem allocator needs to inject cache line flush and memory fence instructions to guarantee crash consistency of heap metadata, this results in a long allocation/release path and significant write amplifications [25]. In addition, the memory requests for large objects in a workload cause lock contention in the allocator. In order to improve efficiency, modern memory allocators adopt a differentiated allocation strategy according to the size of memory objects, i.e., small objects are allocated from the fast path without any thread synchronization overhead, while large objects are allocated from the slow path after acquiring a lock. As will be shown in Sec 4.5, in the worst case, an engine spends about up to 88% CPU cycles on dynamic memory management.

Third, the CoW engine may not be suitable for building PMem-based OLTP systems. When a transaction updates a tuple, the CoW engine first makes a copy of that tuple and then applies the changes to that copy. This approach allows ensuring crash consistency without logging. However, it comes at the cost of more severe write amplification and higher memory allocation overhead [27].

4.4 Tail Latency

Now, we evaluate the tail latency, another important performance metric of the database systems. We collect experimental results under low-skew distribution with 16 threads for YCSB, and use its default configuration for TPC-C. Results are shown in Figure 6 and Figure 7.

4.4.1 YCSB tail latency

For the insert operation (Figure 6(a)), the CoW engine has lower tail latency than the other two engines, and it is particularly noticeable when Makalu is used. This is because the insert operation of the CoW engine is performed in the dirty directory, which never overwrites committed data, eliminating the overhead of recording changes in log entries for recovery. However, as shown in Figure 6(c), the update tail latency of the CoW engine is higher than the InP and LSM engines. This is because it needs to allocate memory for the newly created tuple copy and persist the modified tuple to PMem after the update operation is completed, which is more expensive than writing log entries to PMem.

For the delete operation (Figure 6(b)), we can see that engines using Makalu achieve a tail latency that is 1 to 2 orders of magnitude higher than the counterparts using other

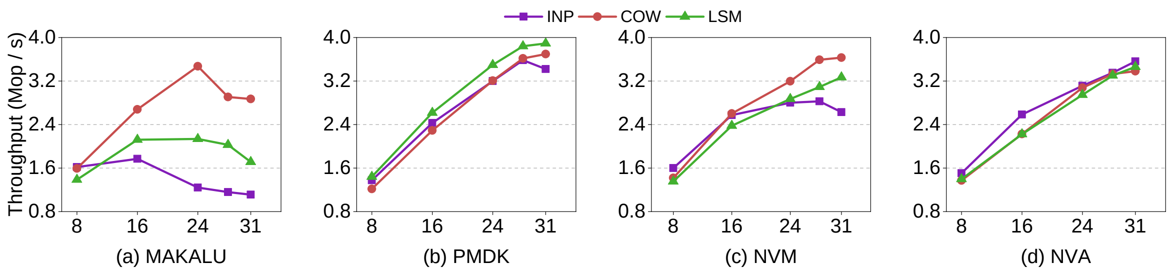


Fig. 4. Performance of variable-length tuples. The workload contains uniformly distributed tuples ranging in size from 1 byte to 256 bytes.

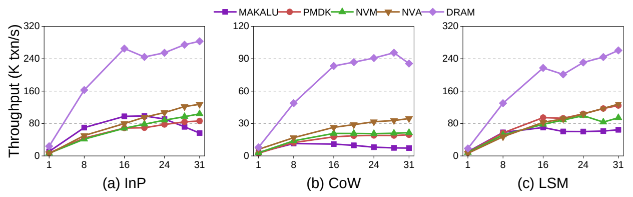


Fig. 5. TPC-C Throughput.

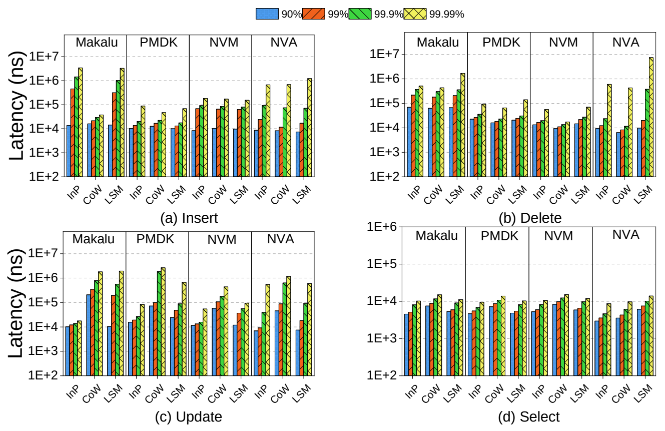


Fig. 6. Tail latency of YCSB.

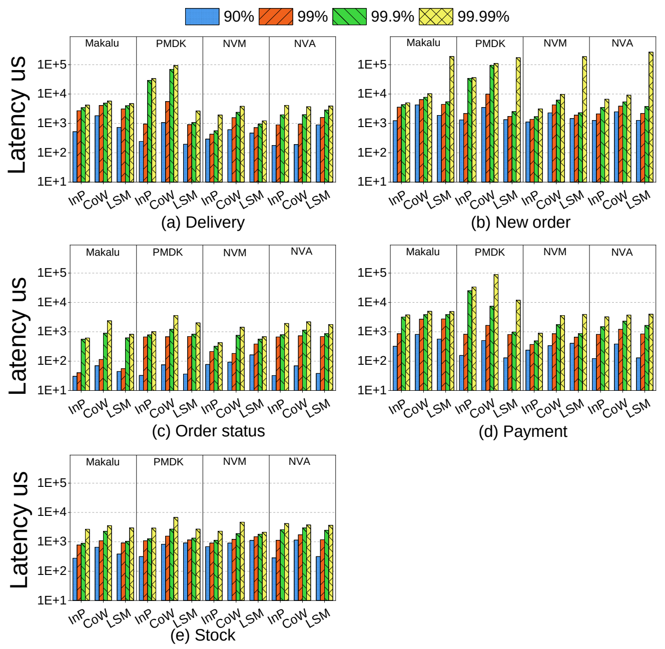


Fig. 7. Tail latency of TPC-C.

allocators. The reason is that Makalu suffers from severe lock contention on the *global reclaim list* when reclaiming the space of deleted tuples, as described in Section 4.3.1. The in-place update policy employed by the InP engine makes it achieve the lowest update tail latency, as exhibited in Figure 6(c). As shown in Figure 6(d), the tail latency of select shows unnoticeable variance for the same engine when different memory allocators are used. In addition, the CoW engine has a higher tail latency than the InP and LSM engines because of its complicated look-up process as mentioned in Section 4.3.1.

4.4.2 TPC-C tail latency.

For write-type transactions as shown in Figure 7(a)(b)(d), the tail latency of PMDK-based engines can be an order of magnitude higher than other cases. The reason for this is twofold. (1) PMDK uses a hybrid undo-redo logging to ensure power-fail atomicity [36], which incurs high flushing overhead. According to prior work [57], with the hybrid redo-undo logging, a transaction involves 4-23 flush instructions and 5-11 fence instructions. (2) To minimize the startup time of the pool, PMDK persists the heap metadata lazily and in stages at runtime, rather than persisting it all at once during initialization [58]. Therefore, although this approach reduces the time required to recover the heap from a system failure, it results in long latency for some memory requests.

The write-type transactions periodically trigger a merge operation in the LSM engine. As shown in Figure 7(b), the tail latency of New Order transactions suffers significantly from the merge operation. Furthermore, no matter which allocator is used, the LSM engine has a tail latency of more than 100 ms at the 99.99 percentile. The reason is that the merge operation in LSM has to access three different tables, whereas for other types of transactions like Payment, it only needs to handle one table.

Compared with YCSB, the tail latency of read-type transactions in TPC-C is much higher (milliseconds vs nanoseconds). This is because the read-type transactions in TPC-C are more complicated. In addition, the overhead caused by computation, intermediate variable generation, and storage during the execution is included when measuring the latency. Moreover, combined with the thread scalability results, we also observe that memory allocators that can obtain high throughput do not always performs well in achieving low tail latency. For instance, the throughput of CoW-P is 1.9x of CoW-M, but the tail latency of the former is 12.1x higher at 99.9 percentile.

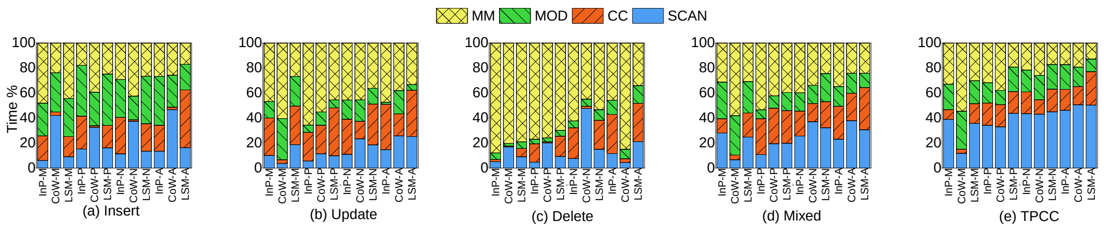


Fig. 8. Execution time breakdown.

4.5 Execution Time Breakdown

We use TPC-C and four YCSB workloads (shown in Figure 8) to analyze the execution time spent on different internal components of each engine. The mixed workload of YCSB is equally partitioned among four operations, i.e., insert, update, delete, and select. For YCSB, we start to measure the execution time after a warm-up phase, in which 2 million records are loaded. Then, 2 million operations are run to measure the execution time. We use 16 threads to run all the experiments.

The execution time is classified into four categories. (1) `SCAN` indicates the overhead of accessing. (2) `CC` means the time consumed in guaranteeing crash consistency, which includes the time of log recording and the execution of flush/fence instructions. (3) `MM` indicates the overhead incurred by PMem memory management. (4) `MOD` presents the execution time of operations that modify the underlying storage engine, such as inserting a new record into a table and deleting a record from a table, and the overhead of maintaining the index data structure, such as inserting/removing nodes in the index structure. As Nstore uses a lightweight concurrency control mechanism, there is no locking or latching overhead at the engine level [10].

Several notable observations can be made from the results shown in Figure 8(a)-(d) for YCSB. First, for the insert operation, `MOD` accounts for 18.9%-41.1% of its execution time, which is more significant than other operations like delete and update. Second, on average, dynamic memory management consumes about 66.3% cycles for the delete workload since the memory space allocated for deleted records needs to be reclaimed. In particular, the percentage of `MM` for InP-M is as high as 87.9%. Third, contrary to our intuition that update operations should have minimal `MM` overhead because they do not need to allocate or reclaim memory space as with the insert and delete operation, we can find from Figure 8(c) that `MM` accounts for a significant portion of execution time as well (45.5% on average). For instance, the `MM` overhead of InP-M, InP-P, InP-N, and InP-A reaches 46.7%, 65.6%, 45.6% and 47.3%, respectively. Similar trends can be observed from the results of the LSM engine. The reason is that before updating the record, the InP and LSM engine need to save old image pointers in a commit vector to ensure data consistency. After the modified data is persisted to PMem, the commit vector is cleared before the transaction is committed. Meanwhile, the space pointed to by the old image pointers is returned to the PMem allocator, which increases the memory management overhead. It is worth noting that the overhead of ensuring crash consistency of the memory heap is not included in `CC`. For the InP and LSM engine, `CC` consists of the overhead of recording log entries for the changes made by transactions and the overhead of persisting updated tuples to PMem using flush+fence instructions. As the CoW engine never overwrites committed data, it does not need to record changes to the tuples in the log file. Therefore, its `CC` overhead is lower than the other two engines, and mainly comes from persisting the contents in the dirty directory to PMem. In the update case (Figure 8(b)), the average `CC` overhead of CoW, InP, and LSM is 14.4%, 29.3%, and 34.6%, respectively. While for insert and delete, the `CC` overhead of the CoW engine is lower (1.8% and 1.6%).

For the TPC-C workload that exhibits similar patterns with the mixed YCSB workload, we have three observations. First, `SCAN` account for more execution time compared to the YCSB workloads (39.7% vs 24.7% on average), reaching 50.6% in the case of CoW-A. This is because performing a TPC-C transaction usually needs to access multiple tables. Second, the LSM engine has the lowest `MM` overhead because it only periodically reclaims the memory occupied by expired log entries. Lastly, except CoW-M, it shows a more regular distribution of execution time among different combinations between storage engines and allocators.

4.6 NUMA Effect

The performance of applications can be heavily impacted by accessing remote NUMA node [25], [59]. And this problem is exacerbated on PMem systems due to the slower access to PMem compared to DRAM and the asymmetric read/write latency, as reported in recent studies [39], [60]. Although several PMem-aware database storage engines have been designed [10], [61], their performance on NUMA systems is under-explored. In this section, we analyze the performance effect of NUMA using YCSB-a, an update-only workload, and the TPC-C workload.

The two CPUs on the test machine are both used in this experiment. One is denoted as local CPU, and the other is denoted as remote CPU. All DRAMs and Optane DCPMMs are installed on the local CPU. We launch 32 thread and pin them to CPU cores with four configurations: (1) `Local` (denoted as `L`) indicates all threads are pinned to the local CPU. (2) `Half` (denoted as `H`) means half of the threads are pinned to the local CPU and the remaining are pinned to the remote CPU. (3) `Remote` (denoted as `R`) means all threads are pinned to the remote CPU. (4) `Default` (denoted as `D`) indicates threads are managed by the default OS scheduler. The results shown in Figure 9 reveal that the read performance of PMem-aware engines is more sensitive to NUMA than the write performance. For example, we can almost double the select throughput (see Figure 9(a)) with the H policy compared to the R policy, while the difference

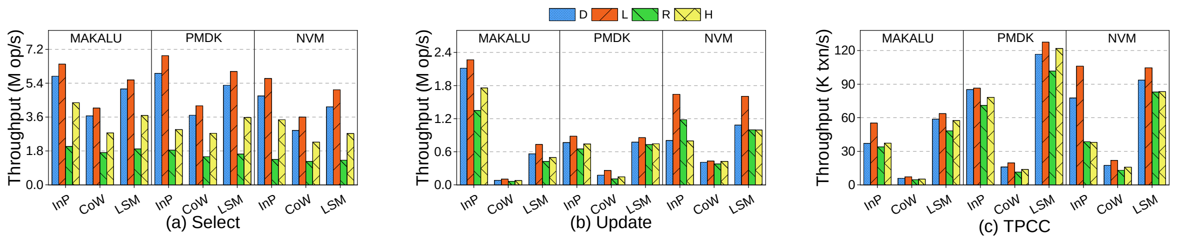


Fig. 9. Throughput comparison using different thread pinning schemes. D represents the default OS scheduler. L means all threads are bound to local CPU. R represents all threads are assigned to the remote CPU. H means threads are equally distributed to the remote and local CPU.

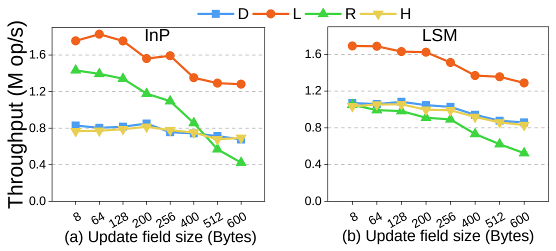


Fig. 10. Throughput of InP-N and LSM-N with different value sizes. All the configurations are the same as in NUMA test except the update field size.

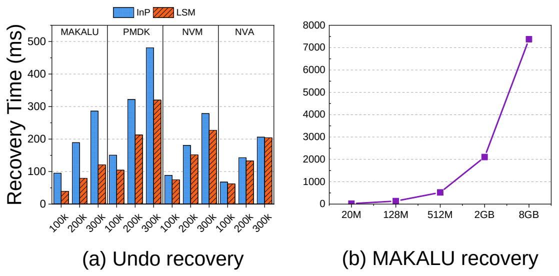


Fig. 11. Recovery (undo) overhead.

of update throughput between the H and R policy is not that prominent (see Figure 9(b)). In addition, the default policy is more friendly to read than write. For example, the D policy greatly outperforms the H policy, as shown in Figure 9(a). However, the update throughput under the D policy is close to that under the H policy as shown in Figure 9(b). A similar observation can be made from the TPC-C workload, as shown in Figure 9(c). This indicates that for read workload, it is more preferable to rely on the default OS scheduler instead of manually pinning threads across different CPUs.

For the update workload shown in Figure 9(b), the L policy shows significant advantage over other policies in only two cases, i.e., InP+NVM and LSM+NVM. Furthermore, it is interesting to observe that the R and H policy achieve similar performance in LSM+NVM, while the R policy greatly outperforms the H policy in InP+NVM. This is because the performance under the R policy is more sensitive to the granularity of data access. As shown in Figure 10, the drop in throughput of the InP-N and LSM-N engine with the R policy is more significant than that with the other policies when the access size varies from 8 bytes to 600 bytes. The results in Figure 9(b) are obtained with the granularity of 200 bytes. Similar trends can be observed for the other allocators.

We make several observations from the TPC-C results shown in Figure 9(c). First, the L policy performs best in most cases, while it only achieves slightly higher performance than the D policy in the case of InP+PMDK. Second, the D policy outperforms the H policy in most cases, except for LSM+PMDK. Third, with nvm_malloc, InP has comparable performance with LSM under the L policy, while it only achieves half the throughput of LSM when the R or H policy is used. In addition, the performance under the R and H policy is close to each other for both InP and LSM. The above analysis implies the complexity in evaluating the performance of PMem-aware storage engines with different NUMA configurations and workloads.

4.7 Recovery

During recovery, the engines will undo the operations in uncommitted transactions according to the log stored on PMem. In order to facilitate the measurement of recovery time, we run a single TPC-C transaction that contains a large number of operations to measure the recovery time, instead of issuing many transactions. In particular, we design a micro-benchmark for this evaluation. It first performs a certain number of operations (including insert, delete and update) within a transaction. Afterward, it skips the commit operation and goes directly to the recovery phase, i.e., undoing the operations executed in the first step. The time elapsed during the undo progress is recorded as the recovery time (in millisecond). In this experiment, we exclude the CoW engine because it does not rely on logging to guarantee consistency.

Figure 11(a) shows the results by running 100K, 200K, and 300K operations within a transaction, from which we can make three observations. First, the recovery time grows linearly as the number of operations in a transaction increases. Second, the recovery overhead of the InP engine is 1.2x to 2.4x higher than that of LSM. This is due to the fact that the LSM engine does not need to access the SSTables in the undo process, even though both InP and LSM are log-based. Third, the PMDK allocator incurs much higher recovery overhead as compared to other allocators. The reclaimed space from the cleaned logs is usually a few bytes in size when rolling back an operation, but even the smallest size-class of PMDK is larger than that. Therefore, the PMDK allocator would spend much more time tracking down these memory objects and returning them to an appropriate freelist.

Next, we discuss the overhead of restoring the heap to a consistent state by the allocators, which is not considered in the above analysis. PMDK, nvm_malloc and NVAlloc maintain bitmaps in their slab headers to keep track of the status of allocated memory blocks. They skip these memory regions by scanning the bitmaps during recovery. In other

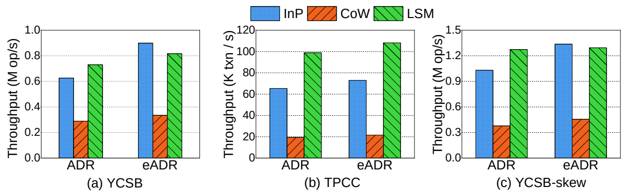


Fig. 12. Performance impact of eADR.

words, they do not perform garbage collection to detect and fix persistent memory leakage in recovery, and thus their recovery time is only several milliseconds. In comparison, Makalu performs garbage collection once the user data in the heap is restored to a consistent state. Therefore, the recovery performance of Makalu depends on the scale of user data. As shown in Figure 11(b), Makalu takes 8000 milliseconds to recover a linked list with 8 GB data.

4.8 eADR Effect

New hardware like the second generation DCPMM with eADR support has been released recently. With eADR, the overhead of PMem write can be reduced because cache-line flushing is no longer needed. To demonstrate the impact of eADR, we measure the throughput of the storage engines when eADR is enabled (eADR mode) and disabled (ADR mode) use three workloads, including (1) a YCSB workload mixed with 90% write (45% insert and 45% update) and 10% read, (2) the TPC-C workload introduced in Section 4.2, and (3) a skewed YCSB workload mixed with 50% update and 50% read. We use the skewed YCSB workload to simulate a situation where there are many reads accessing the just-flushed cache lines. Results are collected with 16 threads on a Dell R750 server with two Intel Xeon scalable processors that support eADR. Under ADR mode, the engines ensure crash consistency by flushing data from the CPU cache to PMem through `clwb` instruction.

Figure 12 shows the results with PMDK. We can find that the throughput is improved when eADR enabled. For YCSB (see Figure 12(a)), the throughput is increased by a factor of 1.44x, 1.12x, and 1.16x for the InP, LSM, and CoW engines, respectively. However, because scanning index takes about 40% of the execution time on average (see Section 4.5) on average for TPC-C, the performance improvement for TPC-C is lower than that for YCSB. As shown in Figure 12(b), with eADR enabled, the throughput of the InP, LSM, and CoW engines is improved by 11.5%, 9.5%, and 10.9%, respectively. In addition, we find that the InP engine is the one that may benefit most from eADR. Although both InP and LSM are log-based, they use different log cleaning mechanisms. InP cleans up logs immediately after a transaction is committed, while LSM cleans up logs periodically when a merging operation is finished. As a result, InP has higher cache flushing overhead than LSM. For skewed YCSB workload, the throughput is increased by a factor of 1.30x, 1.02x, and 1.21x for the InP, LSM, and CoW engines (Figure 12(c)), respectively. The cases with other allocators show similar trends, so we omit the results.

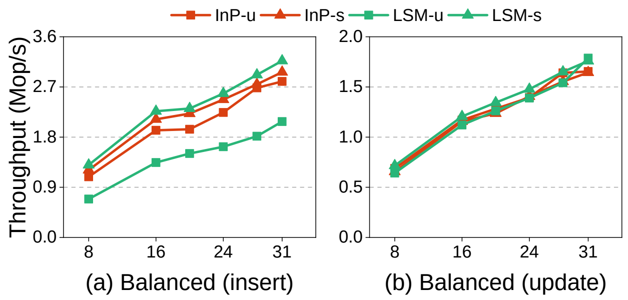


Fig. 13. Throughput of InP and LSM engines (using PMDK) with different B+trees, the suffixes `-u` and `-s` indicate the throughput of the engines when μTree and STXTree are used as the index structures, respectively. Both read-insert balanced (left) and read-update balanced (write) workloads are evaluated.

4.9 Evaluation Using μTree

B+tree and its variants are widely used as the indexes of DBMS systems due to their excellent performance on range scans. For instance, the InP and LSM engines in our benchmark use the B+tree (denoted as STXTree for simplicity) provided by the STX B+tree library as their internal index structures. STXTree is located on PMem and maintained using the PMem allocator interfaces. However, STXTree is not tailored for PMem and lacks crash consistency guarantees. To this end, we may ask what is the impact on performance when the crash consistency of the index is considered. To answer this question, we compare the throughput when using μTree [21] and STXTree as the index, respectively. μTree is a high-performance crash-safe B+tree based on a hybrid PMem+DRAM architecture. The inner nodes of μTree are placed directly in DRAM. It differs from the traditional B+tree in the layout of leaf node. Concretely, the leaf nodes of μTree consist of an array layer residing in DRAM and a list layer residing in PMem. A leaf node in the array layer and list layer are called tree-leaf and list-leaf node, respectively. The former contains multiple key-pointer pairs, and each of them points to a list-leaf node. The list-leaf nodes are organized as a sorted singly-linked list via `next` pointer. This approach enables μTree to achieve fast read performance since look-ups are served in DRAM, and low consistency overhead because it only needs to guarantee the crash consistency of the list layer.

Figure 13 presents the results of the read-insert and read-update balanced workloads. The CoW engine is excluded from this evaluation because it relies on the copy-on-write B+tree to implement shadow paging, a function not supported by μTree. We start by discussing the results of the InP engine on the read-insert balanced workload (Figure 13(a)). In this case, when a tuple is inserted, the InP engine inserts a new entry into the B+tree synchronously. According to the architecture described above, μTree inserts a new list-leaf node in the sorted list layer under the protection of the explicitly injected flush+fence instructions. In contrast, the STXTree appends the entry to a list of entries in the corresponding node directly, which has much lower overhead. Therefore, although μTree has better read performance than STXTree, the throughput of InP-u is slightly lower (less than 10%) than that of InP-s. However, the case of the LSM engine is somewhat different. As shown in Figure 13(a), the performance of LSM-u lags far behind that of LSM-s, with throughput reaching only 61% of that of LSM-s on average. This is because the LSM engine needs to merge a set of SSTables periodically during the process of inserting tuples, a process that results in numerous write operations to the internal index structure. μTree is more affected because each of its write operations is followed by a cache line flush and a memory fence instruction.

When updating a tuple, neither the InP nor the LSM needs to alter the internal index structure, so the performance difference between using μTree and STXTree is small. As shown in Figure 13(b), in this case, the throughput of InP-u is on average 3% higher than that of InP-s, and the throughput of LSM-u is on average 6% lower than that of LSM-s.

5 Discussions

In this section, we summarize the observations and insights made from our evaluation and analysis. Note that none of our insights are specific to Optane DCPMM, except for the last one.

1. Beware of the performance restrictions imposed by random PMem writes. With DRAM-based emulator, it was observed that *the InP engine performs the best across a wide set of workloads* [10]. However, this conclusion is drawn from the incorrect assumption that the performance gap between sequential and random accesses on PMem is comparable to that of DRAM. We find that the performance of the LSM engine, which performs writes sequentially, is clearly better on a real PMem platform. Considering that it is a common feature that sequential I/O is faster than random I/O across different PMem architectures [62], designs that cause fewer random writes or eliminate random writes would be preferred. For instance, leveraging a hybrid PMem+DRAM architecture, the internal index structures of the storage engine can be decoupled from the user data (i.e., the tuples), and random writes generated by updates can be offloaded to DRAM, as in Halo [22] and Viper [46].

2. Interacting with PMem allocator consumes non-negligible overhead. First, it plays a major role in execution efficiency. For the write-intensive workloads (including TPC-C), the engines spend 36.4% (range from 12.8% to 58.0%) of execution time on dynamic memory management on average. In contrast, this overhead on DRAM systems is much lower. For instance, dynamic memory allocation consumes less than 7% of CPU cycles in Google datacenters [32]. Even in read-only scenarios, memory allocation decisions may still impact the performance through data layout. In other words, the throughput and tail latency of the same engine varies with different memory allocators when running the read-only workload. Second, the performance evaluation based on synthetic micro-benchmarks may be misleading when choosing a PMem allocator for real PMem systems. For example, although Makalu achieves higher allocation/deallocation throughput than PMDK in stress testing [53], better performance can be obtained with PMDK on complex workloads in production environments.

3. Tailoring PMem allocators. System designers should be aware of how PMem allocators may affect system scalablity. For example, a PMem allocator should be avoided if it has a lock-protected global structure that can be frequently invoked by system workloads. It is much desired if PMem allocators can be configured to suite application needs with turnable knobs. On the other hand, customizing memory management for specific applications is a good option for achieving maximum performance [22], [63]. For instance, managing PMem with a flat memory space and garbage collection, rather than directly adopting the designs used in traditional allocators.

4. NUMA-awareness is desirable to achieve higher read performance. Remote NUMA access to PMem is slow and has severe negative impact on performance. Considering that the read performance of all engines is more sensitive to NUMA than the write performance, designs that can eliminate remote NUMA read are preferred. For instance, we can adopt the technique similar to the one proposed in [64], which maintains replicas of hot data on each NUMA node to avoid reading data from remote nodes, and synchronizes these replicas via a shared compact log to ensure consistency. This can be integrated into the runtime of a memory allocator [65].

5. No size fits all. According to the experimental results reported in Section 4.3, we can conclude that each engine is unique in its ability of handling particular types of workload. For read-only or read-intensive workloads, the InP engine is preferred, but for update-intensive or insert-intensive workloads, the LSM or CoW engine may be the best choice (depending on which memory allocator is used). The combination of CoW engine and nvm_malloc is optimal when the workload has a high ratio of delete operations [66], [67]. For the TPC-C workload, the LSM engine with PMDK can achieve the best thread scalability.

6. Co-design of PMem-based indexes with real storage systems. Although range indexes designed for PMem systems perform very well when evaluated with synthetic benchmarks, deploying them in storage engines as the internal index structures may not necessarily bring expected performance improvement. For instance, uTree improves the throughput of the InP engine by less than 10%, while it decreases the throughput of the LSM engine by more than 60%. Therefore, we argue that the co-design of the PMem-based indexes with real storage systems must be considered in the future.

7. eADR is not a panacea for better performance. Our evaluation for database storage engines indicates that eADR indeed helps improve performance, but not as much as expected, and it is only beneficial for specific applications and workloads. For instance, when running the write-intensive TPC-C workload, eADR improves the LSM engine's throughput by 9.5%, but it only achieves a 2% improvement with the skewed write-intensive YCSB workload. Given that PMem devices and the microprocessor architectures that support them are in their early stage with rapid evolution, along with the fact eADR-enabled processors are expensive and unlikely to be widely deployed [26], we believe that a system design should not be tightly coupled with features (like eADR) supported by specific hardware. Instead, we should seek for general optimizations that are also valid for future PMem architectures, such as CXL-based PMem [7] and memory semantic SSD.

6 Related work

In this section, we discuss the existing work related to PMem-based storage engines, index structures, and PMem allocators.

PMem-based Storage Engines and Index Structures. DeBrabant et al. [68] explore leveraging persistent memory for online transaction processing (OLTP) DBMS. Leveraging PMem, Yan et al. [69] revisit the conventional LSM-tree based OLTP storage engines originally designed for DRAM-SSD hierarchy, and propose a design over PMem-SSD storage, which outperforms the baseline design by a significant margin. Liu et al. [61] proposes a high-throughput log-free OLTP engine for persistent memory based on hybrid PMem-DRAM architecture, while the storage engines we evaluate in this paper are designed entirely based on PMem. Wu et al. [70] share the lessons learned from the early performance evaluation of Intel Optane DCPMM for DBMS, and conclude that simply changing the hardware or software is not sufficient to exploit the full potential of PMem for DBMS, which is consistent with our observation. The index structure serves as a fundamental component in database systems. In order to facilitate the design and optimization of persistent memory index structures, [25], [27] and [71] have conducted a comprehensive evaluation of the existing PMem-based B+trees and hash indexes with Intel Optane DCPMM, respectively, which are orthogonal to our work.

PMem Allocators. To manage the persistent memory efficiently, a plethora of PMem allocators have been developed in industry and academia [33], [34], [35], [36], [53], [54], [55], [72]. Intel release its Persistent Memory Development Kit (PMDK) [36], which provides a persistent memory allocator. Bhandari et al. [33] propose Makalu, a fast recoverable allocator of persistent memory. Schwalb et al. [34] present nvm_malloc, a general-purpose memory allocator concept with fast built-in recovery features as well as fine-grained access to PMem in a safe and performant fashion. Poseidon [55] provides efficient metadata protection against crashes, API misuse, and memory corruption vulnerabilities by using Intel's Memory Protection key (MPK). But it may incur high PMem footprint due to its use of multilevel hash table for memory management. Ralloc [53] improves Makalu's conservative garbage collector via using user-defined filtering functions to optimize pointer traversal, but it is ill-suited to large object allocations and the complex configuration of database storage engines. For other programming languages besides C/C++, George et al. [72] present a persistent memory allocator in Go. However, there is still a lack of study on how these PMem allocators behave on a real PMem hardware under various database workloads as the above allocators are mostly based on volatile DRAM with simulation. In this paper, we target Makalu, PMDK, nvm_malloc, and NVAlloc, aiming to shed a light on such a picture.

7 Conclusion

This work provides a systematic and comprehensive evaluation of several PMem-aware storage engines on Optane DCPMM-equipped systems. In addition to evaluating how the differences in designs and strategies employed by the engines affect performance, we also consider the intricate impacts of different PMem allocators on the performance of PMem-aware storage engines. We identify key insights for the design of PMem-aware storage engines, PMem allocators, and other correlated components. We hope our observations would be valuable for researchers and practitioners to develop high performance PMem-based systems.

Acknowledgments

This research was supported by the National Science Foundation of China under Grants 61972137 and 61772183, Provincial Natural Science Foundation of Hunan under Grants 2022JJ40088, and Changsha Municipal Natural Science Foundation under Grants kq2202135. The authors would like to thank the anonymous reviewers for their valuable suggestions. We would also like to thank Ms. Jing Liu for helping us to improve this manuscript.

References

[1] 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*, ser. ASPLOS '20. ACM, 2020, p. 1077–1091.

[2] D. Apalkov, A. Khvalkovskiy, S. Watts, V. Nikitin, X. Tang, D. Lottis, K. Moon, X. Luo, E. Chen, A. Ong, A. Driskill-Smith, and M. Krounbi, "Spin-transfer torque magnetic random access memory (stt-mram)," *J. Emerg. Technol. Comput. Syst.*, vol. 9, no. 2, May 2013.

[3] J. J. Yang and R. S. Williams, "Memristive devices in computing system: Promises and challenges," *J. Emerg. Technol. Comput. Syst.*, vol. 9, no. 2, May 2013.

[4] S. Raoux, G. W. Burr, M. J. Breitwisch, C. T. Rettner, Y.-C. Chen, R. M. Shelby, M. Salinga, D. Krebs, S.-H. Chen, H.-L. Lung, and C. H. Lam, "Phase-change random access memory: A scalable technology," *IBM Journal of Research and Development*, vol. 52, no. 4.5, pp. 465–479, 2008.

[5] H.-S. P. Wong, H.-Y. Lee, S. Yu, Y.-S. Chen, Y. Wu, P.-S. Chen, B. Lee, F. T. Chen, and M.-J. Tsai, "Metal–oxide rram," *Proceedings of the IEEE*, vol. 100, no. 6, pp. 1951–1970, 2012.

[6] Micron, "3d-x-point technology." 2015. [Online]. Available: https://www.micron.com/products/advanced-solutions/3d-xpoint-technology

[7] H. Li, D. S. Berger, S. Novakovic, L. Hsu, D. Ernst, P. Zardoshti, M. Shah, S. Rajadnya, S. Lee, I. Agarwal, M. D. Hill, M. Fontoura, and R. Bianchini, "Pond: CXL-Based Memory Pooling Systems for Cloud Platforms," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Vancouver, BC Canada, March 2023.

[8] 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)*. USENIX, Feb. 2020, pp. 169–182.

[9] B. Lu, X. Hao, T. Wang, and E. Lo, "Dash: Scalable hashing on persistent memory," *Proceedings of the VLDB Endowment*, vol. 13, no. 8, pp. 1147–1161, 2020.

[10] J. Arulraj, A. Pavlo, and S. R. Dulloor, "Let's talk about storage & recovery methods for non-volatile memory database systems," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '15. ACM, 2015, p. 707–722.

[11] H. Kimura, "Foedus: Oltp engine for a thousand cores and nvram," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD'15. ACM, 2015, p. 691–706.

[12] F. Xia, D. Jiang, J. Xiong, and N. Sun, "Hikv: A hybrid index key-value store for dram-nvm memory systems," in *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference*, ser. USENIX ATC'17. USENIX, 2017, p. 349–362.

[13] M. Nam, H. Cha, Y. ri Choi, S. H. Noh, and B. Nam, "Write-optimized dynamic hashing for persistent memory," in *17th USENIX Conference on File and Storage Technologies*, ser. FAST'19. USENIX, 2019, pp. 31–44.

[14] P. Zuo, Y. Hua, and J. Wu, “Write-optimized and high-performance hashing index scheme for persistent memory,” in *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'18. Carlsbad, CA, USA: USENIX, 2018, pp. 461–476.

[15] 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*, ser. SIGMOD '16. ACM, 2016, p. 371–386.

[16] J. Arulraj, J. Levandoski, U. F. Minhas, and P.-A. Larson, “Bztree: A high-performance latch-free range index for non-volatile memory,” *Proc. VLDB Endow.*, vol. 11, no. 5, p. 553–565, Jan. 2018.

[17] J. Yang, Q. Wei, C. Chen, C. Wang, K. L. Yong, and B. He, “Nv-tree: Reducing consistency cost for nvm-based single level systems,” in *13th USENIX Conference on File and Storage Technologies (FAST 15)*. USENIX, Feb. 2015, pp. 167–181.

[18] D. Hwang, W.-H. Kim, Y. Won, and B. Nam, “Endurable transient inconsistency in byte-addressable persistent b+-tree,” in *Proceedings of the 16th USENIX Conference on File and Storage Technologies*, ser. FAST'18. USENIX, 2018, p. 187–200.

[19] B. Zhang, S. Zheng, Z. Qi, and L. Huang, “Nbtree: A lock-free pm-friendly persistent b+-tree for eadr-enabled pm systems,” *Proc. VLDB Endow.*, vol. 15, no. 6, p. 1187–1200, feb 2022. [Online]. Available: https://doi.org/10.14778/3514061.3514066

[20] X. Zhou, L. Shou, K. Chen, W. Hu, and G. Chen, “Dptree: Differential indexing for persistent memory,” *Proc. VLDB Endow.*, vol. 13, no. 4, p. 421–434, dec 2019. [Online]. Available: https://doi.org/10.14778/3372716.3372717

[21] Y. Chen, Y. Lu, K. Fang, Q. Wang, and J. Shu, “μTree: A persistent b+-tree with low tail latency,” *Proc. VLDB Endow.*, vol. 13, no. 12, p. 2634–2648, jul 2020. [Online]. Available: https://doi.org/10.14778/3407790.3407850

[22] D. Hu, Z. Chen, W. Che, J. Sun, and H. Chen, “Halo: A hybrid pmem-dram persistent hash index with fast recovery,” in *Proceedings of the 2022 International Conference on Management of Data*, ser. SIGMOD'22. ACM, 2022, p. 1049–1063. [Online]. Available: https://doi.org/10.1145/3514221.3517884

[23] S. Gugnani, A. Kashyap, and X. Lu, “Understanding the idiosyncrasies of real persistent memory,” *Proc. VLDB Endow.*, vol. 14, no. 4, p. 626–639, Dec. 2020.

[24] A. van Renen, L. Vogel, V. Leis, T. Neumann, and A. Kemper, “Building blocks for persistent memory,” *VLDB J.*, vol. 29, pp. 1223–1241, 2020.

[25] D. Hu, Z. Chen, J. Wu, J. Sun, and H. Chen, “Persistent memory hash indexes: An experimental evaluation,” *Proc. VLDB Endow.*, vol. 14, no. 5, p. 785–798, Jan. 2021.

[26] L. Xiang, X. Zhao, J. Rao, S. Jiang, and H. Jiang, “Characterizing the performance of intel optane persistent memory: A close look at its on-dimm buffering,” in *Proceedings of the Seventeenth European Conference on Computer Systems*, ser. EuroSys'22. ACM, 2022, p. 488–505. [Online]. Available: https://doi.org/10.1145/3492321.3519556

[27] L. Lersch, X. Hao, I. Oukid, T. Wang, and T. Willhalm, “Evaluating persistent memory range indexes,” *Proceedings of the VLDB Endowment*, vol. 13, pp. 574–587, 12 2019.

[28] W.-H. Kim, R. M. Krishnan, X. Fu, S. Kashyap, and C. Min, “Pactree: A high performance persistent range index using pac guidelines,” in *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, ser. SOSP'21. New York, NY, USA: ACM, 2021, p. 424–439. [Online]. Available: https://doi.org/10.1145/3477132.3483589

[29] D. Leijen, B. Zorn, and L. de Moura, “Mimalloc: Free list sharding in action,” Microsoft, Tech. Rep. MSR-TR-2019-18, June 2019.

[30] A. Sriraman and A. Dhanotia, “Accelerometer: Understanding acceleration opportunities for data center overheads at hyperscale,” in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS'20. ACM, 2020, p. 733–750.

[31] A. Hunter, C. Kennelly, P. Turner, D. Gove, T. Moseley, and P. Ranganathan, “Beyond malloc efficiency to fleet efficiency: a hugepage-aware memory allocator,” in *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*. USENIX Association, Jul. 2021, pp. 257–273.

[32] S. Kanev, J. P. Darago, K. Hazelwood, P. Ranganathan, T. Moseley, G.-Y. Wei, and D. Brooks, “Profiling a warehouse-scale computer,” in *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, ser. ISCA '15. ACM, 2015, p. 158–169.

[33] K. Bhandari, D. R. Chakrabarti, and H.-J. Boehm, “Makalu: Fast recoverable allocation of non-volatile memory,” *SIGPLAN Not.*, vol. 51, no. 10, p. 677–694, Oct. 2016.

[34] D. Schwalb, T. Berning, M. Faust, M. Dreseler, and H. Plattner, “nvm_malloc: Memory allocation for NVRAM,” in *International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures - ADMS 2015, Kohala Coast, Hawaii, USA, August 31, 2015*, R. Bordawekar, T. Lahiri, B. Gedik, and C. A. Lang, Eds., 2015, pp. 61–72.

[35] Z. Dang, S. He, P. Hong, Z. Li, X. Zhang, X.-H. Sun, and G. Chen, “Nvalloc: Rethinking heap metadata management in persistent memory allocators,” in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS'22. New York, NY, USA: Association for Computing Machinery, 2022, p. 115–127. [Online]. Available: https://doi.org/10.1145/3503222.3507743

[36] Intel, “Persistent memory development kit.” 2021. [Online]. Available: https://pmem.io/

[37] M. Arafa, B. Fahim, S. Kottapalli, A. Kumar, L. P. Looi, S. Mandava, A. Rudoff, I. M. Steiner, B. Valentine, G. Vedaraman, and S. Vora, “Cascade lake: Next generation intel xeon scalable processor,” *IEEE Micro*, vol. 39, no. 2, pp. 29–36, 2019.

[38] Intel, “Cascade lake.” 2019. [Online]. Available: https://www.intel.com/content/www/us/en/design/products-and-solutions/processors-and-chipsets/cascade-lake/2nd-gen-intel-xeon-scalable-processors.html

[39] B. Daase, L. J. Bollmeier, L. Benson, and T. Rabl, “Maximizing persistent memory bandwidth utilization for olap workloads,” in *Proceedings of the 2021 International Conference on Management of Data*, ser. SIGMOD'21. ACM, 2021.

[40] J. Izraelevitz, J. Yang, L. Zhang, J. Kim, X. Liu, A. Memaripour, Y. J. Soh, Z. Wang, Y. Xu, S. R. Dulloor, J. Zhao, and S. Swanson, “Basic performance measurements of the intel optane dc persistent memory module,” 2019.

[41] Intel, “eADR: New opportunities for persistent memory applications,” 2021. [Online]. Available: https://software.intel.com/content/www/us/en/develop/articles\/eadr-new-opportunities-for-persistent-memory-applications.html

[42] B. Di, J. Liu, H. Chen, and D. Li, “Fast, flexible, and comprehensive bug detection for persistent memory programs,” in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS 2021. ACM, 2021, p. 503–516.

[43] J. Ren, J. Zhao, S. Khan, J. Choi, Y. Wu, and O. Mutlu, “Thynvm: Enabling software-transparent crash consistency in persistent memory systems,” in *Proceedings of the 48th International Symposium on Microarchitecture*, ser. MICRO-48. ACM, 2015, p. 672–685.

[44] S. Liu, K. Seemakhupt, Y. Wei, T. Wenisch, A. Kolli, and S. Khan, “Cross-failure bug detection in persistent memory programs,” in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '20. ACM, 2020, p. 1187–1202.

[45] S. Liu, Y. Wei, J. Zhao, A. Kolli, and S. Khan, “Pmtest: A fast and flexible testing framework for persistent memory programs,” in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '19. ACM, 2019, p. 411–425.

[46] L. Benson, H. Makait, and T. Rabl, “Viper: An efficient hybrid pmem-dram key-value store,” *Proceedings of the VLDB Endowment*, vol. 14, no. 9, pp. 1544–1556, 2021.

[47] H.-J. Boehm, “Fast multiprocessor memory allocation and garbage collection,” 2000.

[48] Z. Chen, Y. Huang, B. Ding, and P. Zuo, “Lock-free concurrent level hashing for persistent memory,” in *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, Jul. 2020, pp. 799–812. [Online]. Available: https://www.usenix.org/conference/atc20/presentation/chen

[49] F. D. E. Team, “Rocksdb: A persistent key-value store for flash and ram storage,” 2022. [Online]. Available: https://github.com/pmem/pmem-rocksdb

[50] J. Evans, “A scalable concurrent malloc(3) implementation for freebsd.” 2006. [Online]. Available: https://github.com/jemalloc/jemalloc

[51] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, “Benchmarking cloud serving systems with ycsb,” in *Proceedings of*

the 1st ACM Symposium on Cloud Computing, ser. SoCC '10. ACM, 2010, p. 143–154.

[52] T. T. P. P. Council, "Tpc-c benchmark (revision 5.11.0)," 2010. [Online]. Available: http://www.tpcc.org/tpcc/

[53] W. Cai, H. Wen, H. A. Beadle, C. Kjellqvist, M. Hedayati, and M. L. Scott, "Understanding and optimizing persistent memory allocation," in *Proceedings of the 2020 ACM SIGPLAN International Symposium on Memory Management*, 2020, pp. 60–73.

[54] 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.

[55] A. Demeri, W.-H. Kim, R. M. Krishnan, J. Kim, M. Ismail, and C. Min, "Poseidon: Safe, fast and scalable persistent memory allocator," in *Proceedings of the 21st International Middleware Conference*, 2020, pp. 207–220.

[56] D. E. Difallah, A. Pavlo, C. Curino, and P. Cudre-Mauroux, "Oltp-bench: An extensible testbed for benchmarking relational databases," *Proc. VLDB Endow.*, vol. 7, no. 4, p. 277–288, Dec. 2013.

[57] 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*, ser. ASPLOS '20. ACM, 2020, p. 775–788.

[58] S. Scargall, *PMDK Internals: Important Algorithms and Data Structures*. Apress, 2020, pp. 313–331.

[59] Z. Chen, X. He, J. Sun, H. Chen, and L. He, "Concurrent hash tables on multicore machines: Comparison, evaluation and implications," *Future Generation Computer Systems*, vol. 82, pp. 127–141, 2018.

[60] Q. Wang, Y. Lu, J. Li, and J. Shu, "Nap: A black-box approach to numa-aware persistent memory indexes," in *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*. USENIX Association, Jul. 2021, pp. 93–111.

[61] G. Liu, L. Chen, and S. Chen, "Zen: a high-throughput log-free oltp engine for non-volatile main memory," *Proc. VLDB Endow.*, vol. 14, pp. 835–848, 2021.

[62] S. Gugnani, A. Kashyap, and X. Lu, "Understanding the idiosyncrasies of real persistent memory," *Proceedings of the VLDB Endowment*, vol. 14, no. 4, pp. 626–639, 2020.

[63] K. Wu, J. Ren, I. Peng, and D. Li, "Archtm: Architecture-aware, high performance transaction for persistent memory," in *19th {USENIX} Conference on File and Storage Technologies ({FAST} 21)*, 2021, pp. 141–153.

[64] I. Calciu, S. Sen, M. Balakrishnan, and M. K. Aguilera, "Black-box concurrent data structures for numa architectures," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 207–221. [Online]. Available: https://doi.org/10.1145/3037697.3037721

[65] S. Kaestle, R. Achermann, T. Roscoe, and T. Harris, "Shoal: Smart allocation and replication of memory for parallel programs," in *2015 USENIX Annual Technical Conference (USENIX ATC 15)*. Santa Clara, CA: USENIX Association, Jul. 2015, pp. 263–276. [Online]. Available: https://www.usenix.org/conference/atc15/technical-session/presentation/kaestle

[66] M. Athanassoulis, S. Sarkar, T. I. Papon, Z. Zhu, and D. Staratzis, "Building deletion-compliant data systems," 2022.

[67] S. Sarkar, T. I. Papon, D. Staratzis, and M. Athanassoulis, "Lethe: A tunable delete-aware lsm engine," in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 893–908. [Online]. Available: https://doi.org/10.1145/3318464.3389757

[68] J. A. DeBrabant, J. Arulraj, A. Pavlo, M. Stonebraker, S. B. Zdonik, and S. R. Dulloor, "A prolegomenon on oltp database systems for non-volatile memory," in *ADMS@VLDB*, 2014.

[69] B. Yan, X. Cheng, B. Jiang, S. Chen, C. Shang, J. Wang, K. Huang, X. Yang, W. Cao, and F. Li, "Revisiting the design of lsm-tree based oltp storage engine with persistent memory," *Proc. VLDB Endow.*, vol. 14, pp. 1872–1885, 2021.

[70] Y. Wu, K. Park, R. Sen, B. Kroth, and J. Do, "Lessons learned from the early performance evaluation of intel optane dc persistent memory in dbms," *Proceedings of the 16th International Workshop on Data Management on New Hardware*, 2020.

[71] Y. He, D. Lu, K. Huang, and T. Wang, "Evaluating persistent memory range indexes: Part two," *Proc. VLDB Endow.*, vol. 15, no. 11, p. 2477–2490, sep 2022. [Online]. Available: https://doi.org/10.14778/3551793.3551808

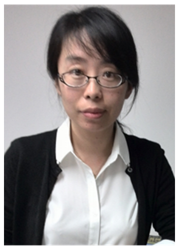
[72] J. S. George, M. Verma, R. Venkatasubramanian, and P. Subrahmanyam, "go-pmem: Native support for programming persistent memory in go," in *USENIX Annual Technical Conference*, 2020.

Zhiwen Chen is a Postdoc at the College of Computer Science and Electronic Engineering, Hunan University, China. He received the Ph.D. degree in Computer Science from Hunan University, China in 2018. His research interests include parallel and distributed computing, operating systems, and storage systems. He has published more than 10 papers in journals and conferences, such as IEEE Transactions on Parallel and Distributed Systems, Future Generation Computer Systems, SIGMOD, VLDB, and ICPP.

Wenkui Che is working toward the Ph.D degree in the College of Computer Science and Electronic Engineering, Hunan University, China. His research interests are persistent memory systems and database systems.

Xin He is a Ph.D student at the College of Computer Science and Electronic Engineering, Hunan University, China. His research interests lie in the intersection of high performance computing and machine learning.

Daokun Hu is working toward the Ph.D degree in the College of Computer Science and Electronic Engineering, Hunan University, China. His research interests include persistent memory systems and database systems.



Jianhua Sun is a Professor at the College of Computer Science and Electronic Engineering, Hunan University, China. She received the Ph.D. degree in Computer Science from Huazhong University of Science and Technology, China in 2005. Her research interests are in security and operating systems. She has published more than 70 papers in journals and conferences, such as IEEE Transactions on Parallel and Distributed Systems, IEEE Transactions on Computers, SIGMOD, VLDB, IPDPS, and ICPP.

Hao Chen received the BS degree in chemical engineering from Sichuan University, China, in 1998, and the PhD degree in computer science from Huazhong University of Science and Technology, China in 2005. He is now a Professor at the College of Computer Science and Electronic Engineering, Hunan University, China. His current research interests include parallel and distributed computing, operating systems, cloud computing and systems security. He has published more than 70 papers in journals and conferences, such as IEEE Transactions on Parallel and Distributed Systems, IEEE Transactions on Computers, ASPLOS, SIGMOD, VLDB, ICS, IPDPS, IWQoS, and ICPP. He is a member of the IEEE and the ACM.