HyperKV: A High Performance Concurrent Key-Value Store for Persistent Memory

Penghao Sun, Dongliang Xue*, Litong You, Yan Yan, Linpeng Huang*
Department of Computer Science and Engineering
Shanghai Jiao Tong University
Shanghai, China
{*sunpenghao, xuedongliang010, litong.you, 118033910141, lphuang*}*@sjtu.edu.cn*

***Abstract*—Concurrency is crucial to achieving good performance in the applications of key-value stores. However, concurrent throughput of traditional key-value stores is inherently limited by the necessity to flush data from DRAM to disks. The emerging persistent memory technology incorporates both byte-addressability and low latency like DRAM, and durability like disks. The characteristics of PM can benefit key-value stores by eliminating the flushing overhead. In this paper, we present HyperKV, a key-value store that leverages PM to provide high concurrent throughput. We achieve concurrency and consistency simultaneously through a fine-grained lock-based concurrency control mechanism of skip list coupled with a novel Action-Oriented Redo Logging (ARL) mechanism. Under ARL, only modification intentions are logged, thereby avoiding write amplification. We apply several optimizations including a hash index in DRAM and a concurrent memory allocator to further improve performance. We evaluate HyperKV against state-of-the-art DRAM-based and PM-based key-value stores on Intel Optane DC Persistent Memory Modules. On YCSB workloads, both throughput and multi-thread speedup of HyperKV are significantly improved compared with NoveLSM and LevelDB.**

***Keywords*-key-value store, concurrency, data consistency, persistent memory, data persistency, memory management**

I. Introduction

Key-value stores are the result of the evolution of storage systems in the era of big data [1]. Key-value stores use a simple data model where data are organized in the form of $\langle key, value \rangle$ pairs [1]. In-memory key-value stores such as LevelDB [2] and Redis [3] are already widely used in applications ranging from web service to high performance computing.

Concurrency is key to achieving good performance in the applications of key-value stores. Researchers have made effort in developing key-value stores with optimized concurrency performance [4], [5], [6]. Concurrency of key-value stores rely largely on the underlying indexes [7], [8], [9], [10]. Although state-of-the-art indexes can offer high concurrency, key-value stores built upon such indexes are inherently limited by the necessity of flushing data from volatile media to non-volatile storage. Implementing persistency mechanisms while maintaining concurrency performance is a challenge. Prior works mainly adopt the log-structured merge (LSM) technique to overcome this chanllenge, but it still damages concurrency performance and leads to write amplification [11].

The emergence of persistent memory (PM) technology is a possible solution to the flushing overhead. PM features byte-addressability and low latency like DRAM and durability like disks [12]. For key-value stores, such characteristics imply that the extra flushing process is no longer necessary. However, using PM for key-value stores introduces the new crash consistency problem. Crash consistency problem refers to possible inconsistency of data in PM after a system crash or power failure. In other words, we need to face the same challenge of ensuring consistency while preserving concurrency. Researchers have designed several PM-based key-value stores with mechanisms to guarantee crash consistency [13], [14], [15], [16]. However, it is observed that their over-sophisticated concurrency control coupled with consistency mechanisms makes it hard to establish their correctness [17].

Motivated by this challenge, we present HyperKV, a high performance key-value store designed to exploit PM and offer high concurrent throughput. We use skip list in HyperKV as the index structure for its simple and scalable concurrency control [8]. Concretely, HyperKV allows lock-free reads and fine-grained lock-based writes where locking is applied on single skip list nodes. We propose a novel logging mechanism for crash consistency, which we name the *Action-Oriented Redo Logging* (ARL). The key idea of ARL is to achieve crash consistency through recording modification intentions. Traditional logging mechanisms duplicate all writes indiscriminately, and crash recovery is achieved by rolling back (undo log) or forward (redo log) partly modified data. In contrast, ARL only persists metadata needed to perform an operation, which can be used to redo actions disrupted by crashes. In this way, write amplification can be reduced. We leverage the hybrid DRAM-PM memory architecture by maintaining a hash index in DRAM, which offers direct access to skip list nodes to accelerate read operations. To further improve concurrency, we also design a concurrent memory allocator for node creation and recycle.

This paper makes the following contributions:

- We present HyperKV, a key-value store architecture that exploits the characteristics of the hybrid memory system by carefully placing metadata and data in DRAM and PM. In this architecture, PM is used to accommodate the index and consistency metadata for immediate persistency, while DRAM keeps other runtime structures to avoid unnecessary PM accesses.

- We propose a mechanism that can ensure crash consistency while preserving concurrency. This mechanism 1) improves concurrent throughput through lock-free reads and fine-grained lock-based writes, 2) achieves crash consistency by logging modification intentions instead of indiscriminate duplication, thereby reducing write amplification, and 3) does not degrade concurrency in the presence of consistency since the logging of intention metadata does not disrupt the locking logic.
- We implement HyperKV and evaluate its performance against state-of-the-art DRAM-based and PM-based key-value stores on Intel Optane DC Persistent Memory Modules. On YCSB workloads, both throughput and multi-thread speedup of HyperKV are significantly improved compared with NoveLSM and LevelDB.

The rest of this paper is organized as follows. In Section 2, we discuss background about key-value stores and persistent memory. Section 3 describes the overall design of HyperKV. Details of our implementation of HyperKV are presented in Section 4. Our evaluation setup and results of HyperKV can be found in Section 5. Section 6 discusses relevant research topics. We conclude this paper in Section 7.

II. Background

A. Key-Value Stores and Persistent Memory

Key-value stores use an extremely simple data model, where data are organized in the form of $\langle key, value \rangle$ pairs. A value is associated with a key, and a key is used to uniquely reference a value. Key-value stores usually keep a large portion of data and the core index structure in main memory for fast insertion and lookup. Typical APIs of a key-value store include:

- **put(*key*, *value*).** Associate a *value* to a *key* and insert the $\langle key, value \rangle$ pair into the key-value store. If *key* already exists, its *value* is overwritten.
- **get(*key*).** Lookup and return the *value* associated with *key*.
- **remove(*key*).** Remove a $\langle key, value \rangle$ pair from the key-value store.

For decades, key-value stores have been developed with the assumption that the system main memory is volatile [1], [2], [3], [4], [5], [6]. The main consideration in this case is to fully exploit the high read and write performance of DRAM. In recent years, however, the situation is changed with the emerging persistent memory technology. PM combines byte-addressability and low latency like DRAM, and durability like disk [12]. It can be placed side-by-side with DRAM as the system main memory. The first and currently only commercial PM product is the Optane DC Persistent Memory [18] from Intel, featuring the 3D Xpoint technology [19].

The motivation of designing key-value stores for PM is clear. First, due to the relatively lower cost per bit of PM, it is possible to accommodate a larger index structure in PM than in DRAM, or even keep the entire data set in PM. This avoids the high performance overhead introduced by expensive I/O operations. Second, initialization of the key-value store system is significantly faster with PM since data structures are stored persistently and available upon startup, whereas in-memory structures must be rebuilt each time the program starts with DRAM. Third, with a well designed consistency mechanism, modifications to data in a key-value store can be persisted immediately without inducing too much overhead. This eliminates the necessity to balance data reliability and I/O overhead due to frequent synchronizations.

B. Persistent Memory Programming Model

The Optane DC Persistent Memory (the Optane memory, here and after used interchangably with PM) operates in two modes, Memory mode and App Direct mode [18]. In Memory mode, the Optane memory is simply treated as volatile memory with no persistence support. When configured in the App Direct mode, the Optane memory is exposed to the operating system as a separate persistent memory device, which can be mounted with a file system. File systems with Direct Access (DAX) support directs read/write requests directly to the underlying device (i.e., PM), bypassing the system page cache [20]. Currently, the ext4 and xfs file systems support DAX and are built-in for Linux. Under this model, data in PM are exposed as memory-mapped files; in other words, applications see PM as chunks of memories in the address space created with the mmap interface.

A crucial problem that must be addressed with PM is crash consistency. Although data that reach PM can survive across power loss, they may be left in an inconsistent state in the event of unexpected system crash or power failure due to processor write reordering and asynchronous cache write back. Therefore, write ordering and cache flushes must be carefully manipulated to ensure crash consistency. Such manipulation is usually achieved with instruction primitives such as mfence and clwb [21].

III. Design of HyperKV

We present HyperKV, a key-value store library for PM. When designing HyperKV, we seek to achieve the following goals: 1) leverage the different performance patterns of DRAM and PM; 2) minimize the performance overhead introduced by consistency mechanisms; and 3) offer high concurrent throughput.

At a high level, HyperKV consists of persistent structures stored in PM and runtime volatile structures stored in DRAM (Fig. 1). To improve concurrency performance, we use

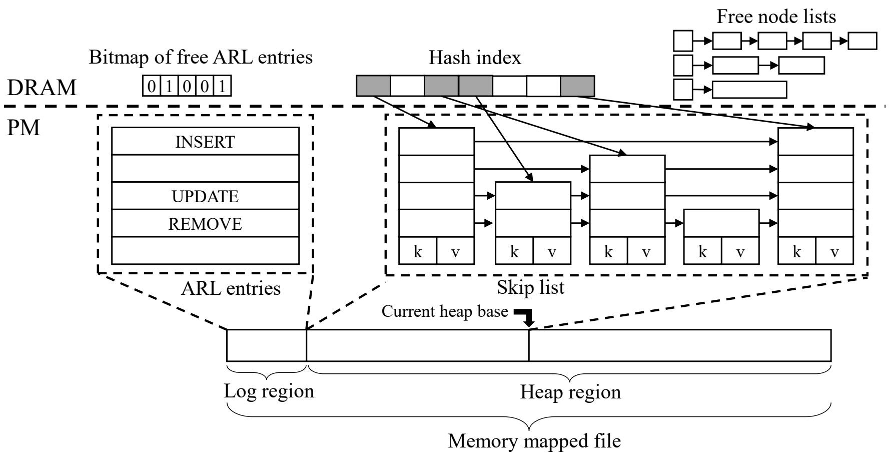

Figure 1. Architecture of HyperKV

Table I
METADATA IN A SKIP LIST NODE

Field	**Type**	**Note**
level	integer	highest level of the node
present	boolean	whether the node is logically present
linked	boolean	whether the node is linked at all levels
lock	lock_t	fine-grained node locking

skip list as the index structure. Nodes of the skip list are dynamically allocated and stored in the heap region of the memory mapped file in PM. To achieve crash consistency, we propose a novel Action-Oriented Redo Logging (ARL) mechanism, where logging is performed on the basis of modification intentions instead of data. All ARL entries are preallocated and stored in the log region. In DRAM, we maintain a hash index offering direct access to nodes in the skip list. Other runtime metadata are also stored in DRAM, including lists of free nodes, the current heap base pointer for new node allocation, and a bitmap for allocating ARL entries. In this way, unnecessary accesses to PM are avoided, which helps save bandwidth and lower access latency.

A. Concurrent Skip List

Skip list is a probabilistic data structure that can achieve O(log$n$) search time and O(log$n$) insert/delete time. Concurrent skip lists can be implemented in a lock-free fashion, allowing high concurrent throughput [7]. This is the key advantage of skip list over tree-based data structures. However, the lock-free concurrency mechanism is too complicated. Achieving crash consistency with minimal cost in this case is impractical. Therefore, we use a lock-based concurrent skip list algorithm which allows lock-free reads and fine-grained locking writes.

Each skip list node contains one $\langle key, value \rangle$ pair. Some metadata are also kept in the skip list nodes (Table I). The *level* field indicates the highest level at which the node is linked in the skip list. The *present* flag is set to false when a thread starts removing the node, but has not fully unlinked the node from the list. In this case, the node is logically removed. The *linked* flag is set to true when the node is linked at all levels. Only after the *linked* flag is set to true are concurrent threads allowed to perform other operations to the node. A *lock* is embedded in each node. To update a node, its *lock* must be acquired first.

With the help of node metadata, concurrent threads performing read or write can determine the logical state of a node, thereby avoiding inconsistency. For example, if a thread starts removing a node but has not finished unlinking the node from all levels, concurrent reads can detect such inconsistent state since the *present* flag will have been set to false. Likewise, if a node insertion has not finished at all levels, a concurrent thread that attempts to remove this node can identify this situation from the *linked* flag. As a result, reads can proceed in a lock-free fashion.

B. Action-Oriented Redo Logging

Existing persistent memory systems employ redo or undo logging mechanisms to guarantee data consistency. However, both redo logging and undo logging involve making extra copies of data, either the modified version or the original version. The extra copy is discarded or overwritten when modifications are committed. For key-value stores, there is space for further optimizations since only a limited number of operations are allowed. Based on this observation, we propose the *Action-Oriented Redo Logging* (ARL) mechanism.

The basic idea of ARL is to log modifications through recording the intention of index modification, or actions, instead of data. For skip lists, possible actions include node insertion, update and remove, which we denote as INSERT, UPDATE and REMOVE, respectively.

Table II
FIELDS OF AN ARL ENTRY

Field	Type	Required for	Note
checksum	integer	INSERT/UPDATE/REMOVE	Used to verify the integrity of the entry during crash recovery
active	boolean	INSERT/UPDATE/REMOVE	Whether the action is ready to be perform
op_type	enumeration	INSERT/UPDATE/REMOVE	Type of the action
value	string	UPDATE	Updated value of the node
node	pointer	INSERT/UPDATE/REMOVE	The node to be inserted, updated or removed
preds	pointer array	INSERT/REMOVE	Predecessors of the node to be inserted or removed

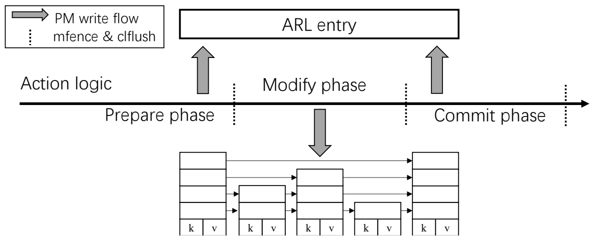

Figure 2. Different phases of an action under the ARL mechanism

Each ARL entry corresponds to an action and contains all necessary information to perform an action. If such information is persisted before data modifications happen, actions disrupted by system crashes can be redone safely. Therefore, we divide the execution of an action into three phases (Fig. 2):

- **Prepare phase.** An ARL entry is allocated. The action logic gathers information needed to perform the action and writes the information to the ARL entry. When all necessary information has been collected and all locks have been acquired, the ARL entry is marked as active.
- **Modify phase.** Actual modifications to the skip list are performed in this phase.
- **Commit phase.** When modifications to the skip list are finished, the action is committed by marking the ARL entry as inactive. The ARL entry is then freed. Node locks are also released in this phase.

Memory fence and cache line flush instructions are executed at the end of each phase to ensure writes during the phase are safely persisted. During recovery from a system crash, actions disrupted in different phases can then be recovered accordingly: 1) For actions disrupted in the Prepare phase, since modifications to the skip list is not ready to be performed, such actions are simply discarded. 2) For actions disrupted in the Modify phase, memory fence and cache line flush ensures that necessary information has been flushed to PM; the actions are redone in this case. 3) For actions disrupted in the Commit phase, it is guaranteed that the action has finished gracefully; but if marking of the ARL entry has not reached PM, the action will be redone. We implement skip list node insertion, deletion and update in an idempotent way to guarantee correctness in this case.

At this stage, the only problem left is to identify which information is needed for each type of action. Take the REMOVE action as an example. To remove a node from the skip list, its predecessors at all levels must be updated to reflect the deletion. Thus, an array of pointers to the predecessors of the node is stored in an ARL entry. The node itself must also be known. Hence, for the REMOVE action, an ARL entry should record the node to be removed and its predecessors. It is worth noting that the process of finding the predecessors of a node is also present in the DRAM version of the algorithm. As a result, keeping such information in ARL entries does not incur overhead in terms of memory writes. More importantly, the original concurrency control mechanism can be preserved because locking of nodes is not affected.

Some other metadata are also kept in ARL entries. For example, a checksum is recorded for integrity check of ARL entries during recovery. The checksum is calculated at the end of the Prepare phase, when all other fields have been set. A complete list of ARL entry fields is provided in Table II. Note that keys and values do not need to be recorded for the INSERT action since keys and values are written to newly allocated nodes, which does not overwrite any existing data. Except for the UPDATE action for which the new value must be recorded, no duplication of data is needed. Consequently, write amplification in HyperKV is minimized.

C. Improving Read Throughput

Skip lists' high insertion and search performance arises from the ability to jump over great numbers of nodes in a single skip. However, wide-span random access can significantly increase access latency of the Optane memory [12]. For HyperKV, this problem is further amplified with concurrent reads and writes. Therefore, measures must be taken to minimize such random accesses of PM.

To do so, we maintain a hash index in DRAM to reduce the number of PM reads needed for a node search. The hash index is essentially an array of direct pointers to skip list nodes in PM. A hash function is used to map nodes with given keys to slots in the index. When a read operation starts, the hash index is checked first. If the corresponding slot of the key contains pointer to the desired node, the process of searching through the skip list can be omitted, thereby

Table III
HyperKV APIs

Name	Input	Output	Description
open	path, size	db	Open or create a HyperKV database and return a reference to it
close	db	N/A	Close a HyperKV database
put	db, key, value	N/A	Insert a $\langle key, value\rangle$ pair into a HyperKV database; overwrite if *key* already exists
get	db, key	value	Get the *value* associated with *key* in a HyperKV database
remove	db, key	N/A	Remove a $\langle key, value\rangle$ from a HypverKV database

Algorithm 1: Allocating a skip list node

```
Input: The level of the node
Output: Pointer to the allocated node
1  freeListIndex = level / 8
2  if !freeLists[freeListIndex].empty() then
3      node = freeLists[freeListIndex].pop()
4      return node
5  end
6  nodeSize = sizeof(struct node) + level * sizeof(struct
     node *)
7  nodeSize = roundUp(nodeSize, 64) // Round up
       node size to multiple of cache
       line size
8  node = currHeapBase
9  currHeapBase += nodeSize
10 return node
```

Algorithm 2: Freeing a skip list node

```
Input: Pointer to the node to be freed
Output: None
1  freeListIndex = node.level / 8
2  freeLists[freeListIndex].push(node)
```

avoiding unnecessary PM accesses. To exploit locality, the hash index is updated after each operation.

IV. Implementation

We implement HyperKV as a C/C++ library and offer a set of APIs (Table III) for programmers to manipulate HyperKV databases. In the remainder of this section, we describe our memory allocator for skip list node creation and recycle. We also provide details of the recovery of HyperKV after a system crash.

A. Memory Allocation

The efficiency of memory allocation is critical to application performance [22]. In HyperKV, the only objects that need to be dynamically created are skip list nodes. Upon creation, level of the node in the skip list is randomly generated. Since only a minor portion of nodes are of higher levels, we use variable size for node structures to reduce memory consumption. To avoid overhead due to cache line misalign, sizes of all nodes are set to be multiples of the cache line size, which is 64 bytes on our machine. Key, value and other metadata take up the size of one or more cache lines, depending on the length of keys and values. Since a pointer is 8 bytes in size, allocated nodes are capable of holding 8 or multiples of 8 pointers. Note that the actual level of nodes may not be multiples of 8 (e.g., level of 5 or 11 is also allowed). The only requirement is that the sizes of nodes be multiples of cache line size.

HyperKV maintains lists of freed nodes in DRAM. Each list contains nodes of the same size; e.g., nodes with 8 pointers are in list 1, and nodes with 16 pointers are in list 2. When a node is allocated, the free list with nodes of the requested size is checked first. If there are available nodes in the list, one is taken and the allocation process finishes. Otherwise, a node is allocated from the heap region. Allocating from the heap can be done by simply updating the current heap base. Concurrent allocations from both free lists and the heap region are lock-free with the help of hardware compare-and-swap routine. On the other hand, freeing a node only involves linking the node to the corresponding free list, which is also implemented without locks. Algorithm 1 and 2 list the pseudocode of node allocating and freeing. Both algorithms can be completed in O(1) time.

B. Crash Recovery

Unfinished index modifications may leave the index structure in an inconsistent state. Therefore, crash recovery must repair such inconsistency. This is achieved by redoing all unfinished actions logged by ARL entries. When a HyperKV database is opened, the recovery program is executed, which checks the checksum of each active ARL entry. If the checksum of an entry matches, its corresponding action is redone using the information contained in the entry:

- **INSERT**: Predecessors of the node to be inserted are recorded in the ARL entry. An INSERT action can be redone by inserting the node after the corresponding predecessor at each level.
- **UPDATE**: Simply updating the node with the new value completes an UPDATE action.
- **REMOVE**: Similar to the INSERT action, a REMOVE action can be redone by unlinking the node from each level by updating its predecessor at the level.

Since the active flag of an ARL entry is set to true only after all locks have been acquired, deadlock is not a concern.

After redoing all unfinished actions, the skip list is restored to a consistent state. Specifically, all removed nodes have been physically deleted from the skip list; i.e., all nodes marked as not present is guaranteed to have been unlinked at all levels. This simplifies the process of heap rebuild. To rebuild heap status, it suffices to set the current heap base pointer to the beginning of unallocated heap region and rebuild free node lists. By zeroing the heap region when a HyperKV database is created, the heap can be rebuilt by traversing through the heap region, finding nodes that are not present and linking them to the free lists, and setting the current heap base pointer to the first node with ”level” 0.

V. Evaluation

In this section, we present experiment results of the performance of HyperKV. When conducting the experiments, we seek to answer the following questions:

1) How well does HyperKV perform in terms of throughput compared to other key-value stores, including those built for traditional DRAM and those specially optimized for PM?
2) How does the relatively poorer performance of PM compared to DRAM affect the throughput of HyperKV?
3) How effective are the optimizations that we have applied to HyperKV, including ARL and the hash index?

A. Setup and Methodology

Our experiments are conducted on a commodity server with 2 Intel Xeon Gold 6240M CPUs, 192 GB DRAM and 1.5 TB Intel Optane DC Persistent Memory. DRAM and PM are equally provisioned to each CPU socket. All Optane memory modules are configured in the App Direct mode and mounted with an ext4-DAX file system. For tests that involve multi-thread operations, we bind all threads to a single NUMA node.

We evaluate the performance of HyperKV and competitors by measuring YCSB workload throughput. Unless explicitly specified, results are measured over 1 million operations (put or get) with 1 million pre-inserted records, and the size of hash index is set to be 1M (2^{20}) slots. All numeric results presented in this section are the average of 10 independent runs.

To answer question 1, we evaluate the performance of HyperKV against LevelDB [2] and NoveLSM [14]. LevelDB is a DRAM-based key-value store developed by Google. It uses log-structured merge trees (LSMs) to offer high throughput by allowing in-memory data access. NoveLSM is a LevelDB implementation optimized for PM. NoveLSM leverages PM's persistence by allowing persistent memtables in PM to be mutable. We choose LevelDB and NoveLSM as our evaluation baseline because they both buffer writes in an in-memory skip list (memtable), which is also used by HyperKV to store data in PM. For fairness of comparison, data files of LevelDB and NoveLSM are stored in PM.

To answer question 2 and 3, we implement several variants of HyperKV by disabling certain features or optimizations and evaluate their performance against the original, full-feature HyperKV. Such HyperKV variants include:

- **HyperKV-dram**. In this variant, ALL data are stored in DRAM. We achieve this by replacing the memory-mapped file in PM with an anonymous region in DRAM. This variant is used to measure the performance degradation caused by PM's lower performance.
- **HyperKV-noflush**. We remove all cache line flush and memory fence instructions. Although data consistency in this case is jeopardized, we are able to evaluate the overhead caused by such expensive instructions.
- **HyperKV-nohash**. We remove the hash index in DRAM and let all operations access the skip list in PM directly. It is used to demonstrate the effectiveness of the hash index to speedup reads.
- **HyperKV-tx**. We replace ARL with transaction APIs offered by PMDK, which internally uses undo logs to guarantee consistency. We use this variant to demonstrate the efficiency of our ARL mechanism.

We also monitor PM write traffic on a byte-accurate basis by leveraging the memory protection mechanism in Linux as in [23] to demonstrate the ability of ARL to reduce write amplification. In addition, we investigate the relationship between the size of the hash index and hash index miss rate for read operations, as well as that between thread number and lock contention for write operations.

B. YCSB Throughput

Fig. 3 shows single- and multi-thread YCSB workload throughput with read/write proportion set to 100/0, 75/25, 50/50, 25/75 and 0/100, respectively. Key distribution is set to uniform. (a)-(e) of Fig. 3 provide comparison among HyperKV, NoveLSM and LevelDB. In single-thread workloads, HyperKV outperforms NoveLSM and LevelDB in the 100% read workload by 2.1x and 3.1x, respectively. When writes are present in the workloads, NoveLSM can benefit from its in-memory skip list for write buffering, whereas HyperKV must insert new records directly to PM. As a result, NoveLSM is able to beat HyperKV by up to 2.0x throughput with a single thread (Fig. 3(e)).

Since NoveLSM uses a thread pool to parallelize search in different levels of the LSM, its multi-thread speedup is inherently bound. In the read-only workload, HyperKV outperforms NoveLSM by up to 8.6x (12 threads); but when the number of threads continues to grow, high latency of random access to PM becomes a bottleneck, and the winning margin of HyperKV shrinks. In write-heavy workloads, memtables in DRAM and PM are filled faster as the number

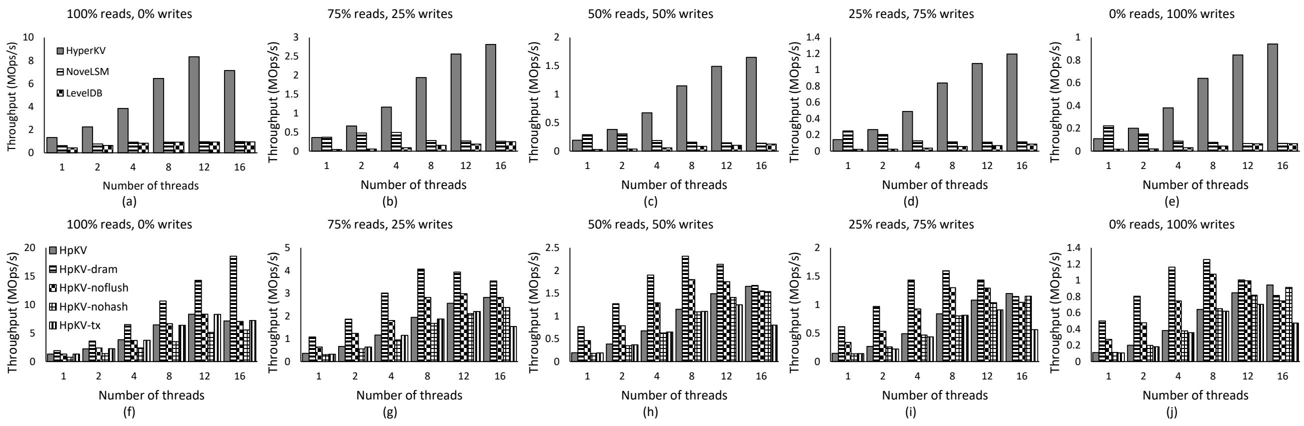

Figure 3. Single- and multi-thread YCSB throughput (uniform key distribution)

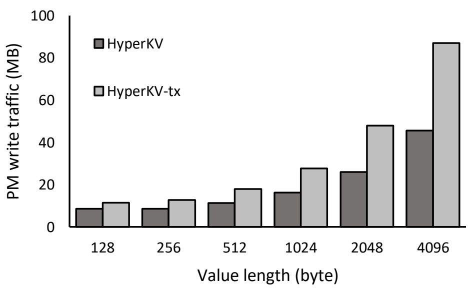

Figure 4. PM write traffic of 10000 insertions

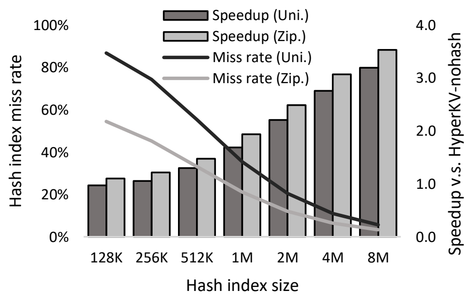

Figure 5. Single-thread hash index miss rate and speedup

of threads increases. This leads to more frequent compactions, thereby limiting throughput. As a result, NoveLSM displays poor multi-thread scalability when writes dominate the workloads. On the other hand, HyperKV scales well with multiple threads thanks to its single level design, resulting in an up to 13.7x performance improvement compared with NoveLSM in the 16-thread write-only workload.

Because LevelDB is designed to use on DRAM coupled with traditional block storage devices, simply putting it in PM fails to utilize PM's potential. Consequently, it is consistently outperformed by HyperKV and NoveLSM.

Fig. 3(f)-(j) shows YCSB workload throughput of HyperKV and all its variants. The following observations are made:

- **HyperKV v.s. HyperKV-dram**: In single-thread workloads, HyperKV-dram achieves consistently higher throughput than HyperKV. The performance gap between the two widens as write proportion increases. Specifically, HyperKV-dram outperforms HyperKV by 1.5x, 2.9x, 3.9x, 4.2x and 4.5x in 0%, 25%, 50%, 75% and 100% write workloads, respectively. This is due to the fact that the current Optane memory model has poorer write bandwidth and latency. As thread number increases, the winning margin of HyperKV-dram grows in read-only workloads: With 16 threads, HyperKV-dram achieves 2.6x throughput compared with HyperKV. However, this trend is reversed when writes dominate the workload due to false sharing. Such problem is most severe in the write-only workload, where HyperKV beats HyperKV-dram by 1.15x with 16 threads.
- **HyperKV v.s. HyperKV-noflush**: HyperKV-noflush beats the HyperKV in write-heavy workloads by up to 2.4x in single-thread workloads, thanks to acceleration of PM access by the CPU cache. However, false sharing is also a problem for HyperKV-noflush in the absence of cache line flush instructions. Consequently, HyperKV-noflush exhibits poorer scalability with multiple threads. Notably, throughput of HyperKV-noflush is only 0.8x of that of HyperKV in the 16-thread write-only workload. Since read-only workload does not require flushes and fences, throughput of the two are similar in all read-only workloads.
- **HyperKV v.s. HyperKV-nohash**: When the hash in-

Table IV
LOCK CONTENTION COUNT, 1M INSERTIONS (AVERAGE OF 10 RUNS)

# threads	Uniform	Zipfian
2	2.2	2.3
4	6.0	6.0
8	13.7	13.9
16	29.5	31.1

dex is removed, read performance is most affected, resulting in 0.58x throughput in the single-thread read-only workload. As write proportion increases, the performance gap between HyperKV-nohash and HyperKV is narrowed. This is because the hash index does not affect the insertion process. Since updating hash slots can be omitted in HyperKV-nohash, it can beat HyperKV by a slight 4% margin in the single-thread, write-only workload.

- **HyperKV v.s. HyperKV-tx**: With a single thread, HyperKV consistently outperforms HyperKV-tx by roughly 5%. This is due to the fact that the space to store undo logs in PMDK is dynamically allocated to suit different log sizes, whereas all ARL entries are preallocated since the size of an ARL entry is predetermined. In addition to undo logs for consistency of program data, PMDK uses separate redo logs for consistency of the memory allocator, which is responsible for allocating space needed for the undo logs. Such sophisticated consistency mechanism can be used for arbitrary data models, but fails to leverage optimization potential in the case of key-value stores. When there are multiple threads, this problem is amplified, resulting in an up to 2.1x performance gap between the two.

C. *Wrtie Amplification Reduction*

Fig. 4 gives the PM write traffic of 10000 insertions for HyperKV and HyperKV-tx. In each test case, value length is fixed. Keys are generated by YCSB and are about 20-byte long each. In HyperKV, the ARL mechanism does not involve key and value copying for insertions; on the other hand, the undo log in HyperKV-tx requires the original content of key and value fields in skip list nodes to be backed up before modifications take place. As a result, HyperKV-tx incurs more PM writes for value lengths ranging from 128 bytes to 4096 bytes. Although ARL can avoid copying keys and values, it does need some extra matadata for redoing actions, resulting in relatively smaller write reduction when values are shorter (24.7% for 128-byte values). However, the size of such metadata does not change with the length of values. Therefore, when values are longer, the effect of write amplification reduction is more significant. Specifically, ARL can reduce PM write traffic by 41.1%, 45.6% and 47.6% for values of 1024-byte, 2048-byte and 4096-byte length, respectively.

D. *Hash Index Sizes*

Fig. 5 shows the relationship between the size of the hash index and its ability to speedup read operations. As the size of hash index increases, miss rate drops steadily, which helps speed up read operations. With a 4M-slot hash index, single-thread read operations are accelerated by up to 3.5x. It is worth pointing out that a 4M-slot hash index only takes up 32MB memory (8 bytes per slot). Such extra space is minor compared to the amount of data usually stored in key-value stores. In addition, for any hash index size, miss rate is lower when key distribution is set to Zipfian than when key distribution is set to Uniform. This is because the Zipfian distribution is skewed and thus has better locality, which results in lower hash index miss rate and higher speedup.

E. *Lock Contention*

Table IV lists lock contention counts of 1 million insertions with varying numbers of threads. The lock contention count is incremented each time a thread tries to acquire a lock that is already held by another thread. From the results we observe that even with 16 threads, lock contention is rare (0.003% of the number of operations). In the skewed Zipfian workload, the probability of multiple threads competing to lock the same node is higher, resulting in marginally more lock acquisition failures. Such results confirm that the fine-grained locking mechanism does not lead to severe lock contention and can improve concurrency.

F. *Discussion*

In this section, we evaluated the performance of HyperKV and competitors by measuring YCSB workload throughput. Our evaluation results confirm that HyperKV can offer high read and write throughput and scales well with multiple threads. The ARL mechanism achieves crash consistency and reduces write amplification compared to traditional logging approaches. A DRAM hash index exploits the hybrid DRAM-PM architecture and improves read throughput at a low spacial cost.

HyperKV also has some limitations. First, to simplify memory allocation, we use fixed-size keys and values in the current implementation. It is possible to store variable length values by managing a dedicated PM region to accommodate values and keeping pointers to values in the skip list nodes, which we did not implement. Second, in terms of consistency, HyperKV only guarantees that ongoing actions that have entered the Modify phase will be safely applied to the database, regardless of system crashes or power failures. If action A starts before action B but when a system crash happens, B has entered the Modify phase while A has not, then only B will be redone during recovery. In other words, there is no guarantee that actions that arrive earlier will be ”safer”.

VI. Related Work

In this section, we briefly summarize recent works on developing key-value stores for PM and concurrency control mechanisms for PM-based data structures.

A. Key-Value Stores for Persistent Memory

There has been a rich body of research on developing key-value stores or indexes for persistent memory. Some focus on converting DRAM key-value stores to PM ones by adding crash consistency support, while other seek to design from scratch. wB$^+$-Tree [13] optimizes the B$^+$-Tree structure for PM by keeping unsorted keys in leaf nodes and using an indirect slot array to record the ordering of keys. In this way, crash consistency can be achieved with atomic writes in most scenarios. HiKV [15] combines the advantage of hash tables and B$^+$-trees to support both fast lookup and ordered index. The hash index is kept in PM, while the B$^+$-tree is stored in DRAM, thereby leveraging the hybrid DRAM-PM memory architecture. SLM-DB [16] uses an LSM-tree to serve as write buffer of a B$^+$-tree. The two structures are kept in PM. SLM-DB also flushes compacted $\langle key, value\rangle$ pairs to disks for larger capacity.

B. Concurrency Control Mechanisms of PM Data Structures

Concurrency control for data structures in PM is complicated due to the necessity of ensuring crash consistency. Nevertheless, researchers have made much effort in this field. FPTree [24] builds a B$^+$-tree with leaf nodes in PM and inner nodes in DRAM. In this way, FPTree is able to use hardware transactional memory for concurrency control of inner nodes. For leaf nodes, fine-grained locking is used. NV-Tree [25] uses multiversion concurrency control (MVCC) by embedding timestamps in the index structure to achieve lock-free concurrent access. Echo [26] is based on a similar technique but features lightweight versioning by leveraging its consistency mechanisms. RECIPE [17] targets DRAM indexes that use atomic writes or visible but reparable inconsistency for isolation. Such indexes can be converted to use PM with a simple, principled approach.

VII. Conclusion

In this paper, we present HyperKV, a key-value store that leverages PM to offer high concurrent throughput. Our Action-Oriented Redo Logging mechanism achieves crash consistency without incurring write amplification or degrading concurrency. To demonstrate the performance of HyperKV, we conduct extensive experiments on Intel Optane DC Persistent Memory Modules. On YCSB workloads, both throughput and multi-thread speedup of HyperKV are significantly improved compared with NoveLSM and LevelDB.

Acknowledgement

We thank the anonymous reviewers for their valuable feedback. This work is sponsored by Natural Science Foundation of Shanghai (No. 21ZR1433600), and SJTU-Huawei Innovation Research Lab Funding (No. FA2018091021-202004).

References

[1] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, “Bigtable: A distributed storage system for structured data,” *ACM Transactions on Computer Systems (TOCS)*, vol. 26, no. 2, pp. 1–26, 2008.

[2] Google, “Leveldb,” https://github.com/google/leveldb.

[3] “Redis,” https://redis.io/.

[4] O. Balmau, R. Guerraoui, V. Trigonakis, and I. Zablotchi, “Flodb: Unlocking memory in persistent key-value stores,” in *Proceedings of the Twelfth European Conference on Computer Systems*, 2017, pp. 80–94.

[5] G. Golan-Gueta, E. Bortnikov, E. Hillel, and I. Keidar, “Scaling concurrent log-structured data stores,” in *Proceedings of the Tenth European Conference on Computer Systems*, 2015, pp. 1–14.

[6] B. Chandramouli, G. Prasaad, D. Kossmann, J. Levandoski, J. Hunter, and M. Barnett, “Faster: A concurrent key-value store with in-place updates,” in *Proceedings of the 2018 International Conference on Management of Data*, 2018, pp. 275–290.

[7] K. Fraser, “Practical lock-freedom,” University of Cambridge, Computer Laboratory, Tech. Rep., 2004.

[8] M. Herlihy, Y. Lev, V. Luchangco, and N. Shavit, “A provably correct scalable concurrent skip list,” in *Conference On Principles of Distributed Systems (OPODIS)*. Citeseer, 2006.

[9] A. Braginsky and E. Petrank, “A lock-free b+ tree,” in *Proceedings of the twenty-fourth annual ACM symposium on Parallelism in algorithms and architectures*, 2012, pp. 58–67.

[10] A. Kogan and E. Petrank, “A methodology for creating fast wait-free data structures,” *ACM SIGPLAN Notices*, vol. 47, no. 8, pp. 141–150, 2012.

[11] C. Luo and M. J. Carey, “Lsm-based storage techniques: a survey,” *The VLDB Journal*, vol. 29, no. 1, pp. 393–418, 2020.

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

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

[14] S. Kannan, N. Bhat, A. Gavrilovska, A. Arpaci-Dusseau, and R. Arpaci-Dusseau, “Redesigning lsms for nonvolatile memory with novelsm,” in *2018 USENIX Annual Technical Conference (USENIXATC 18)*, 2018, pp. 993–1005.

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

[16] O. Kaiyrakhmet, S. Lee, B. Nam, S. H. Noh, and Y.-r. Choi, “Slm-db: single-level key-value store with persistent memory,” in *17th USENIX Conference on File and Storage Technologies (FAST 19)*, 2019, pp. 191–205.

[17] S. K. Lee, J. Mohan, S. Kashyap, T. Kim, and V. Chidambaram, “Recipe: converting concurrent dram indexes to persistent-memory indexes,” in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, 2019, pp. 462–477.

[18] Intel, “Intel optane persistent memory product brief,” https://www.intel.cn/content/www/cn/zh/products/docs/memory-storage/optane-persistent-memory/optane-dc-persistent-memory-brief.html.

[19] Micron, “3d xpoint technology,” https://www.micron.com/products/advanced-solutions/3d-xpoint-technology.

[20] A. Rudoff, “Persistent memory programming,” *Login: The Usenix Magazine*, vol. 42, no. 2, pp. 34–40, 2017.

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

[22] Y. Guo, Q. Zhuge, J. Hu, M. Qiu, and E. H.-M. Sha, “Optimal data allocation for scratch-pad memory on embedded multi-core systems,” in *2011 International Conference on Parallel Processing*. IEEE, 2011, pp. 464–471.

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

[24] 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*, 2016, pp. 371–386.

[25] 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)*, 2015, pp. 167–181.

[26] K. A. Bailey, P. Hornyack, L. Ceze, S. D. Gribble, and H. M. Levy, “Exploring storage class memory with key value stores,” in *Proceedings of the 1st Workshop on Interactions of NVM/FLASH with Operating Systems and Workloads*, 2013, pp. 1–8.