electronics

MDPI

Article

InK: In-Kernel Key-Value Storage with Persistent Memory

Minjong Ha and Sang-Hoon Kim *

Department of Artificial Intelligence, Ajou University, Suwon 16499, Korea; hamj1128@ajou.ac.kr
* Correspondence: sanghoonkim@ajou.ac.kr; Tel.: +82-31-219-3423

Received: 29 September 2020; Accepted: 5 November 2020; Published: 13 November 2020

Abstract: Block-based storage devices exhibit different characteristics from main memory, and applications and systems have been optimized for a long time considering the characteristics in mind. However, emerging non-volatile memory technologies are about to change the situation. Persistent Memory (PM) provides a huge, persistent, and byte-addressable address space to the system, thereby enabling new opportunities for systems software. However, existing applications are usually apt to indirectly utilize PM as a storage device on top of file systems. This makes applications and file systems perform unnecessary operations and amplify I/O traffic, thereby under-utilizing the high performance of PM. In this paper, we make the case for an in-Kernel key-value storage service optimized for PM, called InK. While providing the persistence of data at a high performance, InK considers the characteristics of PM to guarantee the crash consistency. To this end, InK indexes key-value pairs with B+ tree, which is more efficient on PM. We implemented InK based on the Linux kernel and evaluated its performance with Yahoo Cloud Service Benchmark (YCSB) and RocksDB. Evaluation results confirms that InK has advantages over LSM-tree-based key-value store systems in terms of throughput and tail latency.

Keywords: non-volatile memory; key-value store; operating systems

1. Introduction

For a long time, we have been taking computer systems comprised of volatile main memory and non-volatile secondary storage for granted. Those form the memory hierarchy of fast but small DRAM and slow but huge storage devices, and systems software such as operating systems have been designed, implemented, and optimized considering the organization and characteristics of the hierarchy. Meanwhile, there have been constant demands for non-volatile memory for memory-intensive computer systems [1,2]. In response, there have been proposed a number of prototypes and concepts to realize the non-volatile memory. However, the most of the attempts have not been successful yet due to the challenging issues in manufacturing and/or their undesirable characteristics [3–5].

Now the situation is about to be improved by emerging non-volatile memory technologies. In April 2019, Intel released OptaneDC Persistent Memory (DCPM) which is the first commercial product based on a pure persistent memory technology [6]. Based on their patented 3D XPoint technology, DCPM provides the system with a huge, non-volatile, byte-addressable, and over-writable address space. DCPM complies the Dual In-Line Memory Module (DIMM) form factor, so it can easily replace DRAM modules on the memory slots. Due to its high density, one DCPM module can store up to 512 GB of data, which is by an order of magnitude larger than traditional DRAM modules. DCPM exhibits slow but acceptable read/write latency and throughput, and does not wear out by overwrites. All these promising characteristics promote DCPM as the game-changer for modern computer architectures.

Our research has been inspired while evaluating the characteristics of DCPM. Traditionally, storage devices have been complementing the main memory by providing huge, non-volatile spaces to the system. Storage devices have a different access granularity (i.e., block) and characteristics from the main memory, so the processor cannot direct access the data on the storage. This gap necessitates several techniques and concepts, such as file systems, page cache, file-mapped I/O, and prefetching, most of which have been deployed in modern operating systems. However, in spite of the endeavor to bridge the gap, they are becoming the performance bottleneck on high performance storage devices and can even jeopardize the consistency of data [7–10].

Then, why do we need the secondary storage and file systems if the system has a huge, persistent, byte-addressable address space provided by DCPM? Replacing block-addressable storage devices with byte-addressable, non-volatile memory could eliminate these problems originated from the memory hierarchy, and may enable more intuitive, efficient ways of accessing and managing data in computer systems. On the other hand, the system should still provide an alternative to the file systems for the backward compatibility. Usually, applications have been built to store and retrieve data to/from files on file systems. Therefore, to remove the file systems from the system, applications should be changed to adapt to the new concept of stored data. Thus, the next generation computer systems should provide a less intrusive, easy-to-adapt scheme to replace the file system.

In this sense, we attempt to make a case for a key-value store system on non-volatile memory. The key-value store system is an important service component for Internet-scale cloud services nowadays [11–14]. Since it provides a simple yet effective abstract for unstructured data, we counted it as the alternative to the file systems for the next generation computers. However, we discovered that the most popular key-value systems index data through the log-structured merge (LSM) tree [15]. Being optimized for the performance characteristics of block devices, LSM tree outperforms other indexing mechanisms on storage devices in many aspects. However, their operations based on the out-of-place update schemes incurs significant traffic amplification and high performance variation, which are considered as the most critical weakness of the LSM tree. Moreover, their I/O routines are highly dependent to file systems, which we want to remove from the system.

We propose InK, an In-kernel key-value store system optimized for DCPM. Overall, InK is an operating system service that provides user applications with an unified interface to the key-value store. User applications can store a value for a key, and later retrieve it through system calls. We opt to index key-value pairs with B+ tree in that the operations of B+ tree can leverage the byte-addressable capability of DCPM with in-place updates better than the LSM tree. We implemented InK as a subsystem in the Linux kernel. Evaluation using the YCSB benchmark confirms that InK outperforms LSM tree-based key-value store systems by up to 220% and 270% for read and update operations, respectively. Moreover, we can verify consistent I/O performance of InK compared to the key-value store system.

The contributions of this work can be summarized as follows. First, we make a case for the key-value store system in the operating system level. To our best knowledge, this is the first work to make the actual case, and it can enlighten the evolvement of the modern storage system. Second, we report that existing storage systems should be carefully revised to fully leverage the benefit of non-volatile memory. LSM tree has been the most popular data structure for indexing key-value data. However, we found it exposes limitations on non-volatile memory and should be reconsidered accordingly. Finally, we evaluated the system with the industry-standard benchmark.

The rest of this paper is organized as follows. In Section 2, we overview the background and related work of this paper, including B+ tree, LSM tree, and persistent memory. We explain the details of InK design and implementation in Section 3. Section 4 presents the evaluation results of InK. And finally, Section 5 concludes this paper.

2. Background and Related Work

2.1. B+ Tree

B+ tree is one of the most fundamental and popular data structure for indexing items. B+ tree extends the simple binary tree so that each tree node can hold multiple index keys and child nodes. Each index key has two child nodes, and two adjacent index keys share the child node in between. The child node on the left of an index key contains the index keys smaller than the index key. Similarly, the child node on the right contains the index keys larger than or equal to the index key. The maximum number of child nodes in a node is referred to as the order or the fan-out degree of the B+ tree. For example, a B+ tree of order 5 can hold up four index keys and five child nodes in a node.

B+ tree is a balanced tree, so leaf nodes are at the same distance from the root. To maintain the tree balanced, B+ tree may split or merge nodes while inserting or removing items into/from the tree, respectively. To insert a value for a new key in a B+ tree, we need to traverse the tree from the root node to the leaf node by comparing the new key with the index keys in the nodes. When we find a leaf node for the inserting key and the node contains some space for it, we can simply add the key to the leaf node and link the new data to the key. If the leaf node is full (i.e., the number of index keys in the node is equal to the order of the tree minus one), the node should be split to make a space for new keys for subsequent insertion. For this, the leaf node is split into two nodes (say left node and right node), and the left node contains the index keys that are smaller than the index key at the middle of the original node whereas the right node contains the values larger than the middle key. The middle index key is promoted into their parent node and the split nodes are set to the child nodes of the index key in the parent node. The node splitting can take place recursively until a parent node has a room to accommodate the promoted index key without splitting the node. The root node does not have a parent node. When the root node is full, the node is split into two child nodes, and a new node is created as the parent of the nodes. This increases the height of the tree by one, and the root node is replaced with the newly created node.

To remove an item from the tree, we need to first identify the location of the item in the leaf node. If such a leaf node exists, the corresponding index key and value are removed from the leaf node. If the leaf node becomes less than half full by the removal, some index keys and child nodes can be borrowed from its sibling node (i.e., the other child node of the index key in the parent node), or even the entire node can be merged with a sibling node. When a node is completely empty during this processes, it takes index from its parent node to maintaining tree balance. Likewise the node split, this process takes place repeatedly until there are no empty nodes to balancing the tree. If the last remaining index of the root node is taken from the subtree, the height of the tree is reduced by one.

Even the B+ tree is one of the most fundamental data structure for indexing items, there are challenging issues that make it difficult to apply B+ tree to persistent memory. The most challenging issue is the slow write performance. Since the inserting new data to B+ tree takes several steps and changes, its write latency is relatively higher than other indexing data structures such as log-structured merge tree (LSM tree). Thus, even the latency of persistent memory is lower than storage devices, it affects the overall database performance. Concurrency also affects the performance of B+ tree. As described, B+ tree has a worst case scenario that split every nodes in traversal from leaf to root. Without appropriate concurrency policy, it has high limitation for multi-threaded operations. The other issue is inconsistency between CPU cache and persistent memory. Since the consistency between CPU cache and volatile memory is less critical than that between volatile memory and storage, persistent memory has a weak point about in-place update inconsistency. Without solving this problem, we could not use persistent memory like storage device.

There have been extensive work for adapting B+ tree to persistent memory with solving these issues. Hwang et al. [16] observe that updating node data using in-place update such as array shift caused by index inserting, splitting, and merging node has high responsibility to causing flushes between CPU cache and persistent memory. It affects negative on performance, and they proposed

Failure-Atomic ShifT (FAST) and Failure-Atomic In-place Rebalance (FAIR) to solve these problems. FAST is shifting technique to reduce unnecessary CPU cache flushes, and FAIR is an algorithm that performs splitting and merging node efficiently using FAST. Arulraj et al. [17] focus on guaranteeing the concurrency and consistency at the same time. They propose multi-word compare-and-swap operation (PMwCAS) for tree operation that the method of recording the applications of each operation in word on persistent memory and applying them atomically to the tree.

Like the above studies, we designed InK to guarantee the CPU cache consistency and efficient concurrent access using in-place update on persistent memory.

2.2. Log-Structured Merge Tree

Many modern key-value databases, such as LevelDB [11], RocksDB [12], Apache HBase [14], and Cassandra [13], use the Log-structured merge (LSM tree) Tree [15] for its main data structure since it has several advantages. The first advantage is the high write performance than the other data structures. Usually, there are two types of tables for LSM tree, one is so-called memtable which resides in memory, and the other is sstable which stored in storage. Every write and update is firstly recorded in the memtable as a log, and the log is kept until their size reaches a threshold or they pass an expiration time. At the moment, the memtable is flushed to sstable on storage. As this writes are processed, multiple sstables will be accumulated on the storage. And these sstables are continuously merged in the background. Since every write operations occurs in main memory and takes simple steps, it shows high write performance than the other data structures.

Another advantage of the LSM tree is that the LSM tree is optimized for the access pattern following Zipfian distribution which is quietly common in practice. Since LSM tree places frequently accessed data close to the memtable and rarely accessed data to the lower of the sstables, it shows high performance on the workload with bimodal access pattern. More the accessed data pattern follows bimodal, this tendency becomes strong.

However, there are some limitations that using LSM tree based key-value database directly on persistent memory. First, LSM tree has high write amplification since it keep flushes and merges the logs to storage, and maintains write-ahead log (WAL) files for failure recovery. It could be critical issue on persistent memory since the system has limited capacity for persistent memory than traditional storage, such as HDD and SSD. Moreover, since the LSM tree based key-value databases are optimized for volatile memory and non-volatile storage, there are some unnecessary operations for persistent memory. For example, creating WAL files for failure recovery is not persistent memory aware operation. If the memtable is located in persistent memory, it could perform the role of WAL either. Thus, LSM tree needs to be optimized for persistent memory for efficiency.

Kaiyrakhme et al. [18] propose Single Level key-value store (SLM-DB) to optimize LSM tree to persistent memory. They design SLM-DB has the memtable located in persistent memory, and using it as the role of WAL either. Thus, they could reduce the write amplification caused by WAL file on persistent memory. They also make SLM-DB has single level for sstable, and indexing the key-value pair in sstable to B+ tree in persistent memory for fast read performance. Since key-value pairs organized in single level, and these pairs are indexed by B+ tree in persistent memory, they could reduce write and read amplifications. Liu et al. [19] propose NVLevel, redesigning the LevelDB [11] for persistent memory. They divide the one big memtable in persistent memory to multiple memtables, and it makes NVLevel could perform concurrent write operations. They also use skip-list based on hashes for read operation. Since the persistent memory has huge and byte-addressable address, they could implement these approaches on persistent memory, which demands huge capacities.

2.3. Persistent Memory

Persistent memory is the new media that located between DRAM and storage. It connects to the memory bus, and provides accesses at the byte granularity to any address for read/write like DRAM.

However, unlike DRAM loses every data it has after power cycle, persistent memory keeps the data before the power cycle.

A number of technologies have been proposed to realize the persistent memory and to deal with challenges [3–5,20,21]. Intel Optane DC Persistent Memory (DCPM) is the first commercially available persistent memory released in April 2019 [6]. DCPM exhibits 2–3.7× higher read latency with 1/3 bandwidth, and the same write latency with 1/6 bandwidth compared to DRAM. A single system can have up to 6 TB of DCPM. DCPM supports two modes; memory mode and app direct mode. Memory mode uses the existing DRAM as a cache and the DCPM as the main memory. And app direct mode allows the DCPM to be used as a storage device.

To provide these semantics, Linux uses the Direct Access (DAX) feature that maps physical DCPM region into the virtual address space of an application. Since the DAX feature, DCPM requires explicit CPU cache line write backs to ensure the persistence of a modification [22]. It means that if the system failure occurs while the system does not flush the data in CPU cache for DCPM, that data could be loss and disable to recovery. Thus, Intel CPU instructions support flushing of cache line and non-temporal stores: `clwb`, `clflush`, `clflushopt` [23].

Since DCPM has new position in memory hierarchy, there have been studies to exploit its features and performance. Even when DCPM was not available, people tried to provide reliability and offer better performance on persistent memory [1,2,24,25]. Dulloor et al. [25] propose PMFS, which is the file system that optimize for byte-addressable storage using memory-mapped I/O (mmap). Since the traditional file system is only optimized for block device, they focus on the file system for persistent memory. Volos et al. [2] present Mnemosyne, which is a simple programming interface with persistent memory. Mnemosyne could creates and manages the objects on persistent memory with consistency and failure recovery. Thus, programmers could utilize persistent objects like managing volatile memory objects. Coburn et al. [24] also propose NV-Heaps, the object manager for persistent memory. They focus on preventing the familiar bugs such as multiple free()s, locking errors, and a new bug that only appeared in persistent memory, hard-to-find pointer safety bug. Hard-to-find pointer safety bug is caused when non-volatile data structure points the volatile memory object. Since this pointer becomes meaningless after the program end, the system should distinguish where the objects belong. NV-Heaps implement three pointer types to prevent this bug: Non-Volatile (NV)-Volatile (V) pointer, NV-NV pointer, and weak NV-NV pointer. Venkataraman et al. [26] focus on the problem that traditional data stores have: updates in-memory data and sync the data to storage. Since these traditional data stores are not assumed for persistent memory feature, updating the same data twice in memory and storage is huge performance overhead for persistent memory. Instead, they propose Consistent and Durable Data Structure (CDDS) based on B- tree with atomic updates using `clflush` instruction. Chen et al. [27] studies persistent in-memory B+ tree since it is widely used in database. However, B+ tree operations such as inserting and deleting index from node requires movement of index array, and it invokes the extensive persistent memory write and CPU cache flushing operations causing drastic performance overhead. Thus, they propose wB+ tree. wB+ tree uses unsorted indexes to reducing the index movement, and provides atomic writes to guarantee the data consistency between CPU cache and persistent memory. pmemkv is a key-value data store optimized for persistent memory [28]. It provides several storage engines that using the same API [29], and these engines have different data structures such as hash map, B+ tree, and radix tree. Main difference between InK and pmemkv is that pmemkv is in user space, and InK works in kernel space. Another difference is B+ tree based pmemkv engines do not support concurrent executions, while InK supports it.

Compared to those previous studies, InK is unique in that it provides the full system-level service in the operating system. Nevertheless, InK considers those issues discussed in the previous work.

3. InK Design

This section overviews InK first, and then explains how InK handles various key-value operation types and concerns. To fully take the advantages of the byte-addressable and over-writable address space, we designed InK based on the B+ tree, and optimized for efficient concurrent accesses. At the same time, we considered the characteristics of the non-volatile memory device to guarantee the crash consistency of stored key-value data.

Figure 1 illustrates the overall architecture of InK . InK provides a number of system calls that correspond to the key-value operation types that InK supports. An application can invoke the system call to access, insert, modify, and delete the key-value pairs in InK . The system call requests are forwarded to the InK submodule in the kernel, and are processed through a B+ tree that indexes the address of each key-value pair on the DCPM. Finally, the process result is returned to the requesting application through the traditional system call interfaces.

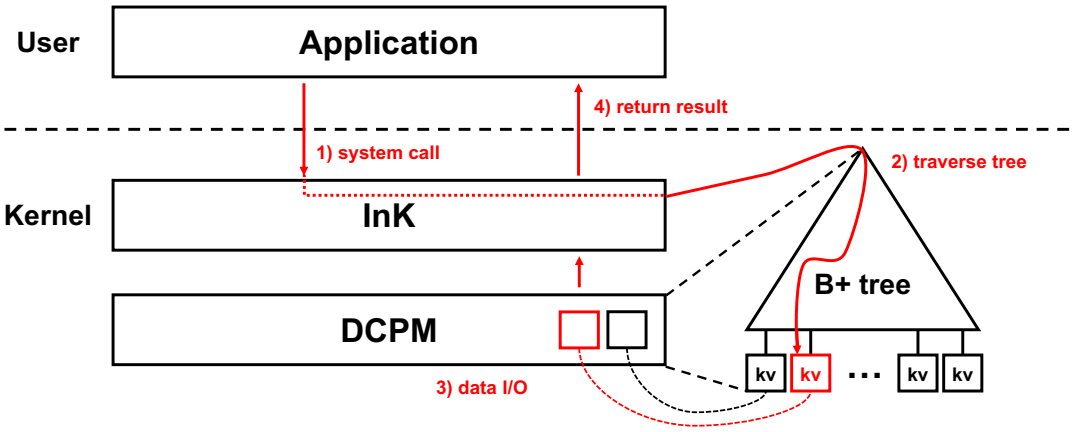


Figure 1. The overall architecture of InK system.

3.1. Managing Address Space on DCPM

DCPM provides two modes for utilizing its persistent address space. One of the modes is Memory mode, in which DCPM provides the system with the vast address space on DCPM as regular memory. The system can utilize DCPM as a large, energy-efficient main memory module without modification. However, the data stored in the Memory mode is not persistent (i.e., destroyed upon a power cycle), and the memory access latency varies when the internal memory controller (IMC) which controls DCPM migrates the data between DCPM and the main memory back and forth [6].

The other mode is so-called App Direct mode [6]. IMC provides the system with a separate, persistent address range that the system can explicitly manage. In practice, the Linux kernel manages the persistent address range as similar to the device-mapped memory region; the address range exists, but the system does not utilize the range for serving virtual memory of the processes. Instead, the address range should be mapped to an address space, and then accessed using general memory instructions such as load and store.

InK utilizes DCPM in the App Direct mode. InK maps the entire memory region of DCPM into the kernel address space. The mapping address of the DCPM memory region is not fixed on a particular address but can be changed on a system reboot. Thus, InK represents data in DCPM with the relative address to the start of the mapped address; InK calculates the location by adding the starting address and the offset, and directly accesses to the target address through general memory instructions. Since DCPM allows in-place updates, modification can be applied directly to the address space.

InK partitions the non-volatile address space on DCPM into three areas, and Figure 2 illustrates those areas. The first area fixed at the beginning of the address space of DCPM is called metadata area, and it stores the metadata of InK such as the address of the B+ tree root node, the information for managing DCPM address space, the descriptor for logging area, so forth. During the system initialization, InK reconstructs the respective in-memory data structures by reading the data from the metadata area. Moreover, InK uses the logging descriptor to identify an unexpected crash of the system.

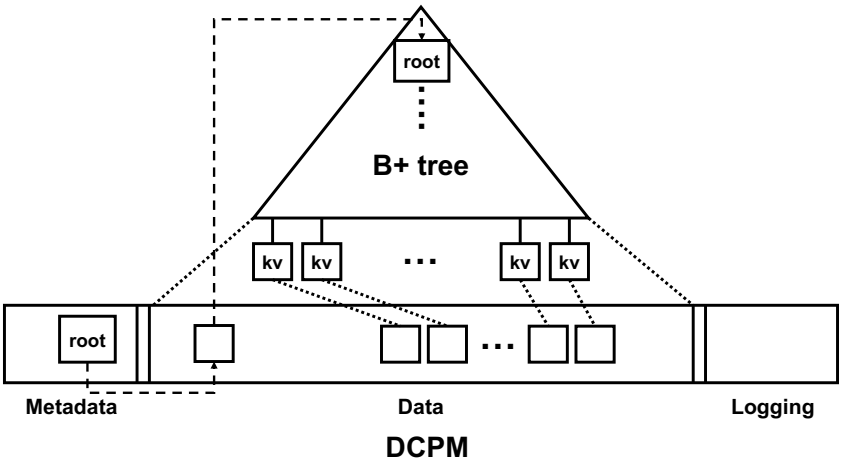


Figure 2. The address space of DCPM partitioned for InK.

The second area is for storing the actual data of key-value pairs and their index. It comes right after the metadata area and occupies the majority of the space on DCPM. Basically, keys and values are stored in this area as a contiguous chunks. To accelerate the lookup for stored key-value pairs, InK maintains a B+ tree instance in this area. The tree indexes all key-value pairs in InK. The leaf nodes of the tree is comprised of index keys and value pointers and they store the locations of the chunks for actual keys and values. The usage of the area is managed by the space allocation information stored in the metadata area (see Section 3.4).

The last area is for the logging changes in InK. Internally, The changes of the B+ tree indexes and key-value data are updated in place on the persistent address space on DCPM. This in-place update approach is a double-edge sword; it allows InK to leverage the byte-addressable capability of DCPM, but it complicates the case when the updates are interrupted in the middle of operation. Thus, it is essential for InK to detect such a crash and recover from the transient, inconsistent state. Combined with the logging descriptor in the metadata area, InK detects such an inconsistent state and then recovers from it (details are discussed in Section 3.5).

The size of the areas can be set while initializing the InK instance.

3.2. Indexing Key-Value Pairs

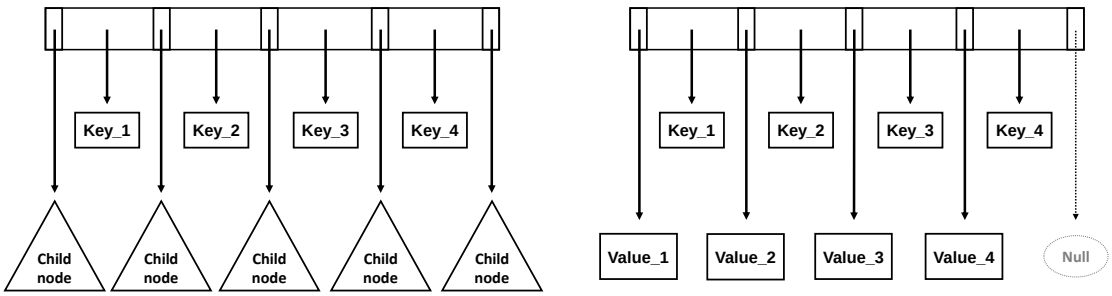
Many state-of-the-art key-value store systems [11–14] internally index the key-value pairs with the log-structured merge (LSM) tree [15]. As we explained in Section 2.2, LSM tree can maximize the I/O performance by leveraging the superior sequential I/O performance of block devices. Such a LSM tree has, however, inherent shortcomings to be integrated on top of the byte-addressable and over-writable address space [30].

First, the LSM tree operations are designed by assuming the I/Os at a block granularity. Small updates are inevitably merged in the memory (i.e., in memtable), and are written to storage devices at the block granularity (i.e., in sstables). Reads from the storage devices are also performed at the same block granularity. This amplifies the read and write traffic, and also slows down I/O.

Second, LSM tree is designed to leverage many performance characteristics of block devices, however, many of them do not hold anymore on the byte-addressable non-volatile memory. For instance, LSM tree performs out-of-place update to leverage the sequential write performance. However, random writes to DCPM shows comparable performance to sequential writes in terms of latency. Updates can be applied directly to DCPM, allowing small in-place updates. Moreover, the lifespan is not as significant issue as it does on the flash memory-based devices. Third, reads from LSM tree exhibits a long tail latency. Since items are stored in a hierarchical order, LSM tree need to scan many sstables to access cold items. This incurs multiple accesses to the storage device extends read time. Lastly, the compaction has been problematic in LSM tree. The compaction is the process of cleaning out-dated/deleted data in LSM tree [31]. To do that, LSM tree reads sstables, merges them, and produces sstables. This process generates heavy I/O traffic in the background [32,33], interfering foreground key-value services.

Based on these observations, we opt to implement InK based on the B+ tree. As one of the most traditional and popular in-memory data structures for indexing items, B+ tree does not have these shortcomings. As the balanced tree, all items in a B+ tree are at the same distance from the root node, thereby providing consistent and bounded lookup time. Since the DCPM address space can be directly accessed through load and store instructions, we can access and update key-value data in-place without amplifying I/O traffic. Moreover, B+ tree can continue operating without depending on critical background processes.

Figure 3 illustrates the layout of index nodes and leaf nodes of the B+ tree with order 5. The index node has five pointers pointing to five child nodes, and four pointers are placed between these child nodes. The pointers store the location of the index keys (thus one index node is effectively comprised of nine pointers). The index key is not inlined in the node but separately stored in the memory. This layout is not generally used on block-based devices since such an indirection causes additional I/Os to access keys, significantly degrading overall performance. This indirection is, however, not so costly on the byte-addressable DCPM, and even helps InK to use the large fan-out degrees of tree nodes.



(**a**) Index node (**b**) Leaf node

Figure 3. The layout of tree nodes in the B+ tree index.

Leaf nodes are organized in the similar way. It contains eight pointers; four of them are for pointing to the address of values and four of them are to the keys. The value pointer on the left of an index key points to the location of the value of the corresponding index key. The rightmost pointer is always set to null in the leaf nodes.

3.3. Handling Basic Key-Value Operations

InK maintains a B+ tree for indexing the key-value pairs stored on the non-volatile address space. When an application queries for the value for a key through an InK system call, InK looks up the

requested key from the B+ tree index. Starting from the root node, InK finds the proper index key and subtree in the node. InK uses the binary search technique to accelerate the search for the index key and subtree in the nodes. If InK finds a proper subtree from the node, it iterates to search the requested key from the pointed subtree. This procedure is repeated until the search reaches a leaf node. If the leaf node contains the requested key, InK returns the value that the value pointer in the leaf node points to. If the key is not in the leaf node, it implies that the requested key does not exist in InK. Thus, InK returns an error code (`-ENOENT`) indicating the situation.

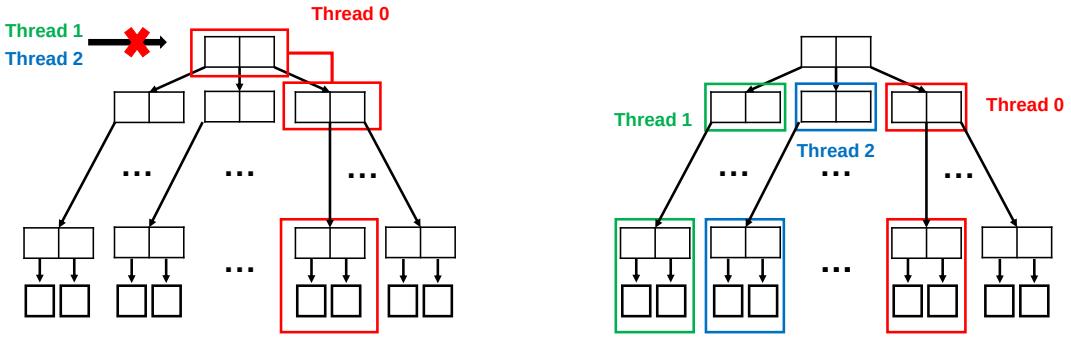
When a write for a key is requested, InK starts traversing the index tree as similar to the key-value lookup. When it reaches a leaf node and the requested key does not exist in the leaf node, it implies that the key-value pair for the requested key does not exist in InK. In this case, InK allocates memory chunks for the key and value from the data area (see Section 3.4 for space management), copies the requested key and value onto the allocated chunks, and then inserts them into the appropriate position in the leaf node. To keep the index keys in the node sorted, InK finds the position for the new key through the binary search, and then shifts the index key and value pointers from that position, and then puts the newly allocated key-value pair at the location. If the key for the write request exists in the tree index, InK replaces the old value with the new value in the request. If the size of the value is changed, InK allocates a new memory chunk for the updated value, fills it with the new value, and replaces the value pointer of the key. The memory chunk storing the previous value is reclaimed through the memory management mechanism of InK. When the size of the value is not changed, InK just copies the new value to the existing chunk. Note that the leaf node containing the requested key is not changed for updating a key-value pair with the same-sized value. Thus, InK can omit logging the leaf node. Since the value size for a key does not change frequently [34,35], this approach effectively improves the overall performance of InK.

It is crucial for a key-value store system to effectively control the concurrent accesses to the stored key-value pairs so that the applications using the system can maximize the aggregated I/O throughput. This necessitates an efficient control for the concurrent accesses. Basically, InK uses the fine-grained reader-writer locks (i.e., `rw_semaphore` in the Linux kernel). Each B+ tree node contains a reader-writer lock instance that serializes the concurrent accesses to the data in the node. In general, the context accessing the B+ tree index starts the access from the root node, and moves towards leaf nodes comparing the key with the index keys. InK starts accessing the index keys by grabbing the lock of the root node. For the operations that only read the index, InK locks the current node with the reader lock, allowing multiple readers to access the index concurrently. When the requested operation may alter the index, InK grabs the writer lock whereas it takes the reader lock for read-only operations. To descend down to the leaf nodes, InK grabs the same type of the lock on the next level while holding the lock of the current node. The lock for the current node is released when it is guaranteed that further operation processing does not modify the node. This way, multiple get requests can be simultaneously processed whereas write requests only lock a part of the index, thereby maximizing the concurrent access to the index.

Figure 4 shows the benefit of the fine-grained locking mechanism for a B+ tree. In the coarse-grained lock mechanism, only one thread can access the tree and other threads are blocked by the lock of the root node. In contrast, the fine-grained lockng mechanism allows multiple threads to access the each parts of the tree concurrently.

Applying the proposing locking mechanism in B+ tree seems straightforward. However, controlling concurrent access in B+ tree becomes challenging when a node needs to be split and even increases the tree height. When a leaf node becomes full, the node needs to be split to add more keys that falls into the index key range the node covers. The index key at the middle of the current node is inserted into the parent node of the current node. The current node is split into two nodes, so that one of the nodes contains the index keys that are smaller than the middle index key, whereas the other node contains the rest. Those split nodes are attached to the left and the right of the index key that is newly inserted into the parent node. Since the split inserts an index key to the

parent node, the internal index nodes (i.e., parent node) can get full as well as the leaf node, triggering the split of the index node. Thus, the node split firstly take place at a leaf node, and then is recursively propagated towards the root node as long as the current node becomes full. However, this direction of the node split, climbing up from leaf nodes towards the root node, is opposite to the direction of the request processing, which descends down from the root node to leaf nodes, complicating the concurrent access control in B+ tree. Worse, since the node split modifies at least three nodes (current node, newly allocated node, and parent node), the split operation should be controlled so that other contexts do not access the nodes that are in transition.



(**a**) Coarse-grained locking mechanism (**b**) Proposing fine-grained locking mechanism

Figure 4. The benefit of the fine-grained locking mechanism of InK.

To handle the node split efficiently, we propose a lazy split scheme in the B+ tree. The key idea is to postpone the split of full nodes until it is accessed again. In the original B+ tree, a full node is split immediately when it gets full. Thus, a node split at a leaf node can trigger a chain of node splitting up to the root node. In contrast, InK splits the nodes in a lazy way. When a node becomes full, InK leaves the node unchanged until the node is to be traversed again. As stated above, InK grabs the lock of the child node while holding the lock of the current node to traverse the tree from the root node to leaf nodes. If the child node is not full, InK releases the lock of the current node, and continues searching the target index key. If the child node is full, InK splits the node while holding the locks of the current node and the child node. The middle index key of the child node is inserted into the current node, and the child node is split into the two child nodes (in fact, InK makes two child nodes by allocating a new node and moving a half of the index keys in the original node to the new node). All nodes influenced by the split is protected by the locks, therefore, the split can be performed without causing a concurrency issue. After the split, InK releases the lock for the current node, and continues traversing the child node. This approach unifies the node access directions of normal operations and node splits, thereby simplifying the concurrency control. Moreover, one write operation can split only up to one node, therefore, it can contribute to the steady performance for write operations.

In practice, the majority of key-value operations in key-value stores is comprised of get, put, and update operations, and deletion is an uncommon operation [34]. We designed InK to handle the deletion in an optimistic and lazy way and focused on optimizing common cases faster. When an application requests InK for deleting a key-value pair, InK finds the corresponding index key and value pointer from the B+ tree index. If such a key-value pair exists in the leaf node of B+ tree, InK reclaims the spaces storing the actual key and value of the pair. The pointers in the corresponding leaf node are also modified. The update only takes place at the leaf node, and index nodes are not modified to remove key-value pairs. Moreover, InK does not actively merge the nodes with low utilization unlike the original B+ tree. Instead, InK can merge low utilization nodes on background, which can be triggered during an idle period of the system. Since this operation is not critical to the correctness of the system, InK can perform the operation only if the system is idle and it may not disturb the system

performance. As it is known that storage systems have a plentiful amount of idle time, we believe this approach might not harm the overall system performance in InK.

3.4. Managing DCPM Address Space

The memory manager in InK manages the key-value area on the DCPM address space, which stores actual key-value data and B+ tree nodes. Many studies analyzing key-value store systems in action commonly report that the majority of keys and values are very small, and their sizes are highly skewed to only a few sizes [34,35]. If InK manages the space at a large, fixed-sized granularity, like pages or blocks, InK will suffer from high internal fragmentation since small keys and values are common. On the other hand, employing general, sophisticated memory management schemes in the kernel space can increase the code base size, which is not desirable for the stability and security of operating systems. Moreover, that way is unable to exploit the characteristics of key-value data. We designed the memory manager considering these characteristics and design constraints so that InK can manage the huge non-volatile address space at a low overhead.

The memory manager manages address space of DCPM with two data structures, allocation pointer and free space lists. Figure 5 illustrates the key mechanisms that InK employs to manage the address space. The allocation pointer points to the address of a free memory chunk. Initially, this pointer points to the start address of the key-value area in Figure 5. InK may allocate a space from the address that the allocation pointer points to. After allocating the space, the allocation pointer is increased by the allocated size, so that it always points to the start address of the free space.

The free space lists is an array of lists, as illustrated in Figure 5. The number of list array entries is 10 by default, which is configurable during the InK instance initialization. Each list links the free space chunks whose sizes are 2 to the power of 4 plus the index of the list. For example, the third list links free chunks of $2^{(4+2)} = 64$ bytes. A pointer is embedded at the beginning of each free chunk, which points to the next free chunk available. When InK needs a free chunks, InK looks up the free space list that corresponds to the requested size. When a free chunk is available in the corresponding free space list, it is detached from the list. If there is no free chunk corresponds to the requested size, InK allocates a new chunk from the allocation pointer as we explained above. When InK frees up a space, it is attached to the head of the corresponding free space list.

Our evaluation on memory management shows this approach inevitably incurs external fragmentation. However, due to the workload characteristics of key-values, the external fragmentation has been remained at a manageable level. Specifically, reclaimed spaces are immediately reused for subsequent requests in the same size, thereby leaving less than 0.2% of space in the free list throughout the evaluation.

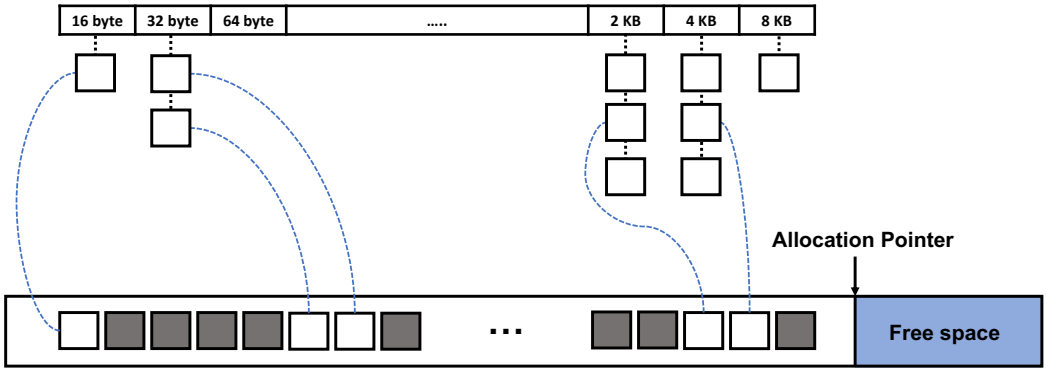


Figure 5. Managing address space in the data area.

3.5. Consistency

Unlike flash memory, DCPM allows in-place update and InK leverages this feature. However, DCPM only guarantees the atomicity of operations at the cache line granularity, and the orders between cache lines nor the atomicity for a large area are not guaranteed. This complicates the situation when a B+ tree node modification in InK is interrupted in the middle of the operation by the system crash. Suppose a B+ tree node became full and is about to be split by the lazy split scheme. The split requires to insert a newly allocated index key into the current node, and to populate two nodes each of which contains a half of the index keys in the original child node. To keep the B+ tree consistent and durable, these changes should be atomically applied to the B+ tree index. However, DCPM architecture does not provide the way to atomically update those multiple locations.

To provide the required consistency and durability on the DCPM address space, InK employs the logging technique with in-place updates. Before applying a modification to the key-value data or B+ tree nodes, InK copies the original data to the logging area on the DCPM address space. The copied data includes the allocation pointer, free space list for memory management, the previous key-value data in the chunks, and the data in the tree nodes. InK maintains the logging descriptor which is stored in the metadata area. The logging descriptor initially points to the start address of the logging area, and then is adjusted to point to the end of ongoing log. InK copies the data to preserve to the location where the logging descriptor points to, and then moves the logging descriptor to the end of the copied data accordingly. Only after copying all the data to be changed, InK starts applying the changes to their original location. When the changes are completely applied, InK resets the logging descriptor to the beginning of the logging area. Whenever the value of the logging descriptor is changed, the value is persisted to the metadata area. This implies that there was an unexpected interrupt while applying some changes in InK if the logging descriptor does not point to the start address of the logging area. In this case, InK recovers the inconsistent state by restoring the original value from the log. After restoring the original values, InK resets the logging descriptor to indicate InK is fully recovered. InK also can recover from another failure during the recovery since copying the original value again is an idempotent operation.

When InK writes data on the DCPM address space, InK should consider the side effect of cache in the processors. Specifically, due to the memory hierarchy of the systems, updates to memory is applied to the cache which is volatile, and then later written back to non-volatile DCPM. This may cause a partial update to DCPM address space, leading the system to an inconsistent state [36]. InK prevents this happening by leveraging the architecture support. Intel introduces `clwb` instruction, which enforces the processor to flush the specified cache line into the memory [37]. When InK updates data on DCPM address space, InK ensures that updates are committed to DCPM by invoking `clwb` for the updated memory range.

4. Evaluation

This section reports the evaluation results of InK. We firstly analyze the performance characteristics of InK, and then compare its overall performance to RocksDB. The evaluation is performed on a server which is equipped with an Intel Xeon Gold 5215 CPU running at 2.50 GHz. The server is equipped with two 16 GB DDR4 DRAM DIMMs (32 GB of memory in total), and one 128 GB Intel Optane DC Persistent Memory (DCPM) module. As we previously explained, DCPM is set to the App Direct mode to explicitly manage the non-volatile address space. As the benchmark, we used the Yahoo Cloud Service Benchmark(YCSB) [38,39] with default parameters unless otherwise specified.

We implemented InK based on the Linux kernel v5.0.3. InK implementation introduces four system calls to user applications, each of which corresponds to the basic key-value operation of key-value store systems. The system calls are summarized in Table 1. To perform the evaluation with YCSB, we built a binding to convert the key-value operations generated by YCSB to corresponding system calls. The implementation took around 1400 lines of code, and we are planning to make the code public to contribute to the system software research.

Table 1. InK system calls for key-value operations.

Function Description
`int ink_init(erase)`
Initialize the InK instance on the system. Clear all existing data if `erase` is true.
`int ink_insert(key, key_size, value, value_size)`
Add the key-value pair. Update the value if the key already exists.
`int ink_read(key, key_size, value, *value_size)`
Copy the value of the `key` to `value` and set `value_size` accordingly.
`int ink_remove(key, key_size)`
Remove the specified key-value pair from tree.

4.1. InK Performance Analysis

First, we analyze the performance of InK on various fan-out degrees (i.e., order) since it determines the overall performance of the B+ tree index. On each run, we initialized a new InK instance with 100 million key-value pairs, and then measured the performance of the next 100 million key-value operations. Read and update operations are generated to the keys used for initializing the InK instance. For write operations, the benchmark generates write requests to non-existing keys so that the operations add new key-value pairs into the InK instance. These keys are selected to conform to the Zipfian distribution with the parameter $z = 0.99$. All keys and values are 23 bytes and 100 bytes in size, respectively, which is the default setting of YCSB.

Figure 6 summarizes the throughput of InK when a single thread performs the operations. Overall, InK shows better read and update performance with high order configurations. Specifically, InK handles update and read operations at 102,009 and 145,357 operations per second when the order is 7, respectively, and 124,114 and 180,442 operations per second with order 255. This is mainly due to the time complexity of B+ tree. If the tree is not modified, the performance is solely determined by the time to look up the key, which is bounded to $O(log_n N)$ where n is the order of nodes and N is the number of keys in the tree. Thus, the higher order the InK is configured with, the better InK performs.

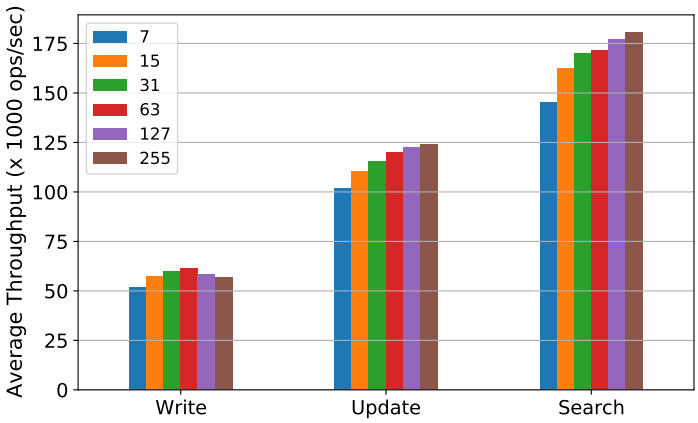


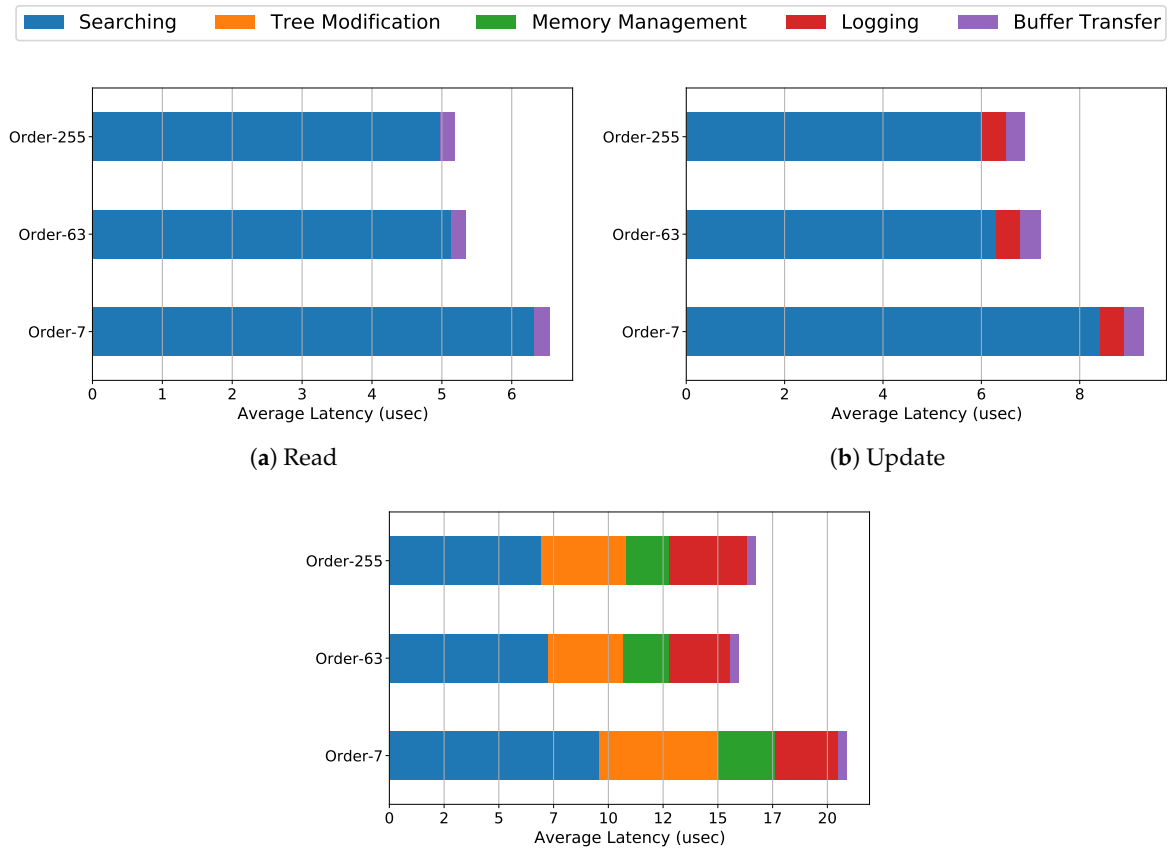
Figure 6. The throughput of the InK key-value operations.

The write operations show a different performance trend. The performance is increased up to order 63, and then slightly decreased on larger orders. We attribute this trend to the overhead coming from other than the key lookup. Unlike the read and update operations, write operations modify multiple locations on the tree to insert the new key-value pair and even to split full nodes. This takes place along with memory allocation and logging.

To quantify those overhead, we broke down the latency for each operation type. We broke the average latency into following categories.

- Searching: the time to lookup the keys from the index tree
- Tree Modification: the time to add or update the tree node.
- Memory Management: the time to allocate or return the space from/to DCPM address space
- Logging: the time to process the log for the system consistency
- Buffer Transfer: time to transfer the key-value data between buffers in the user- and kernel-space

Figure 7 summarizes the breakdown result. Read operations do not modify the tree nor allocate memory. Thus, searching takes the majority of the time, and the order of B+ tree is the only parameter that affects the searching performance Figure 7a. We can observe consistent or negligible time from other categories. Update operations show the similar trend to the read operation. Searching takes slightly longer than the read operations since InK should descend the locks with write permission for update operations. The updated value is overwritten to the original buffer, and InK makes a log for the update to cope with an unexpected interrupt of the in-place update. This logging incurs additional overhead to the searching.



(**a**) Read (**b**) Update

(**c**) Write

Figure 7. The breakdown of the latency for each key-value operation type.

Write operations are handled through many steps. Unlike other operation types, tree modification, memory management, and logging take significantly longer time. To insert a new key-value pair, the space for the key and value should be allocated from the DCPM address space, and then the key is inserted into the leaf node of the index tree. The key insertion may also trigger a split in the subsequent write operations. This modifies the tree node, and the metadata for the memory management. When InK uses a small order, the nodes will be split frequently. On the other hand, tree update takes longer on a large order to insert a new index key into the node, since a new key insertion is processed by shifting index keys in the nodes. The more data to be updated, the more time the logging takes. Overall, the time is reduced up to order 63, and then slightly increased afterwards.

Thus, InK performs write operations best at order 63, and the rest of the evaluation has been done with the configuration.

4.2. Performance on Various Workloads

We compare the performance of InK with RocksDB [12], which is one of the most popular key-value store systems. Since RocksDB is designed to run on top of a file system, it cannot directly utilize the DCPM address space. Thus, we set up a file system (ext4) on DCPM, and configured RocksDB to use it for storing data. We evaluated the performance of RocksDB with the DAX option on and off to see the performance implication of DAX. For a fair comparison, we enabled write-ahead logging (WAL) on RocksDB so that RocksDB guarantee the same consistency level of InK. We used the same configurations that we used in Section 4.1.

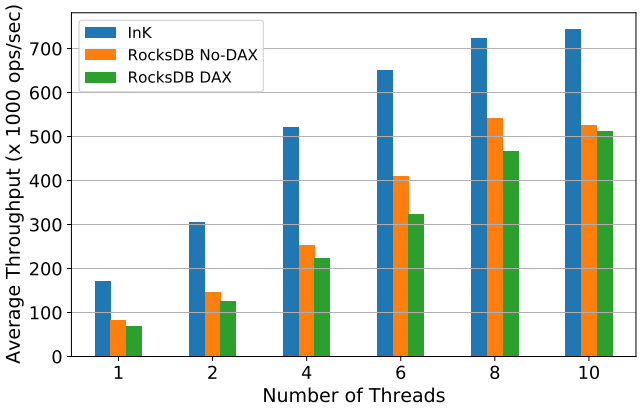
Figure 8 compares the performance of each operation type on InK, RocksDB with DAX, and RocksDB without DAX. We measured the aggregated throughput while increasing the accessing threads from 1 to 10. Overall, both RocksDB and InK show scalable performance; the aggregated throughput is increased when more threads are accessing the system. InK outperforms RocksDB for read and update operations. InK outperforms RocksDB by up to 220% and 270% for read and update operations, respectively. However, RocksDB outperforms InK for write operations by up to 270%. RocksDB is optimized for fast write; it maintains recent writes in the main memory as memtable, and then later flushes the memtable to the storage, minimizing the overhead for write operations. However, this complicates read and update operations; RocksDB may search multiple levels of sstables to lookup the target key. Due to the performance characteristics, we can conclude that RocksDB better suits for write-intensive workloads whereas InK is for read- and update-intensive workloads. Note that it has been reported that the majority of key-value operations are for read and update operations and write and deletion are not common in general [34,35,39]. Thus, we carefully propose that InK may have some opportunities to improve the system performance in these cases.

From all configurations, we can verify that RocksDB runs slower when the DAX option is set. This is counter-intuitive in that DAX eliminates the memory accesses to the page cache. We attribute the slowdown to the slow performance of DCPM than DRAM. With DAX enabled, the updates to key-value pairs are directly applied to DCPM without going through the page cache. Although the memory accesses to the page cache is eliminated, each operation takes longer since DCPM operates slower than DRAM. This can increase the overall process time for the workload with a locality in references. We can learn a lesson from the result that the characteristics of NVM and the workload should be considered together for maximizing the system performance, otherwise the system may under-perform.

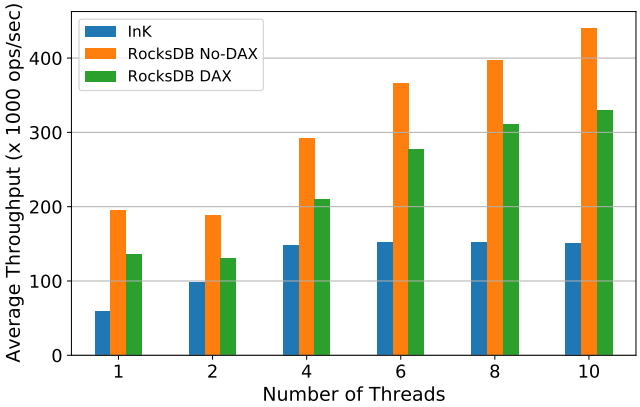
There have been projects to design key-value store systems considering the characteristics of non-volatile memory, and we compared the performance of InK to pmemkv [28]. Pmemkv is designed for modularity, so it affords various language bindings and storage engines. The storage engines include concurrent hash map, volatile sorted hash map, volatile concurrent hash map, radix tree, and so on. Among these storage engines, only the concurrent hash map (cmap) supports data persistency and concurrent data access at least as InK does. However, cmap only allows the concurrent data access within the process boundary, not from multiple processes, thereby making a fair comparison impossible. In addition, there was an issue in the stock Java binding that is limiting the maximum size of the store instance to 2 GB. Thus, we can only evaluate the single threaded performance from a small-sized instance with 450,000 key-value pairs. Note that InK is built as the kernel service, so it allows parallel accesses from multiple multi-threaded processes and is capable of utilizing the entire persistent address space.

Figure 9 summarizes the performance of InK and pmemkv, showing that InK outperforms pmemkv for all operation types. In particular, InK shows lower latency than pmemkv for write, update, and read operations by 11.7%, 30.3%, and 26.9%, respectively. Based on the result, we can estimate the performance of the systems with a large configuration. The latencies in Figure 7 are

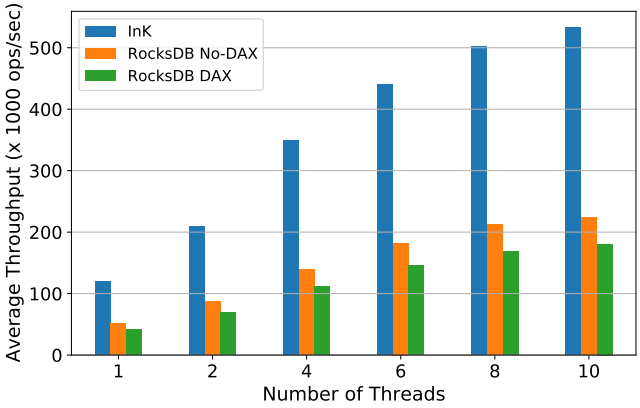
obtained when InK manages 100 million key-value pairs, but they are close to those shown in Figure 9. This implies that InK shows consistent performance with respect to the number of stored key-value pairs. In contrast, the performance of pmemkv will be degraded on a large configuration since the hash map, the mechanism to index key-value pairs in cmap, is subject to experience more collisions that extend the processing time. Thus, the performance gap between InK and pmemkv will be widen on large configurations, and we can carefully suggest that InK is better than (or at least comparable to) pmemkv in terms of performance.



(**a**) Read



(**b**) Write



(**c**) Update

Figure 8. Aggregated throughput on multithreaded configurations.

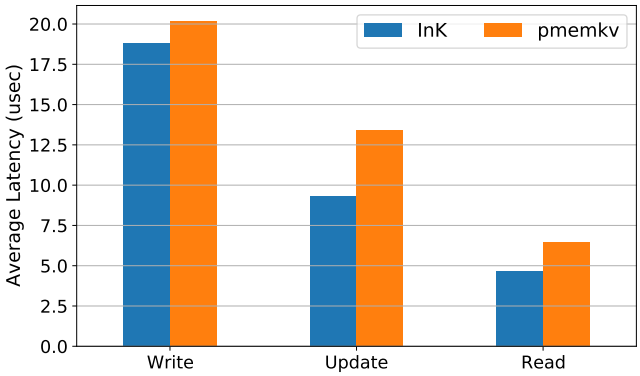


Figure 9. The basic performance of InK and pmemkv.

We evaluate the performance of RocksDB and InK on YCSB with realistic workloads. We initialized the system with 100 million key-value pairs with the keys and values are 23 bytes 100 bytes in size, respectively. Then, YCSB is configured to perform 100 million key-value operations, which are in the ratio summarized in Table 2. We evaluate the performance with two different locality of references conditions; when the keys are selected by the uniform random distribution (without a locality) and by the Zipfian distribution with 0.99 as the parameter (with a high locality). Figure 10 summarizes the result.

Table 2. The operation composition of Yahoo Cloud Service Benchmark (YCSB) workloads.

Workload	Update	Read	Description
Workload A	0.5	0.5	Update-intensive
Workload B	0.05	0.95	Read-intensive
Workload C	0.0	1.00	Read-only

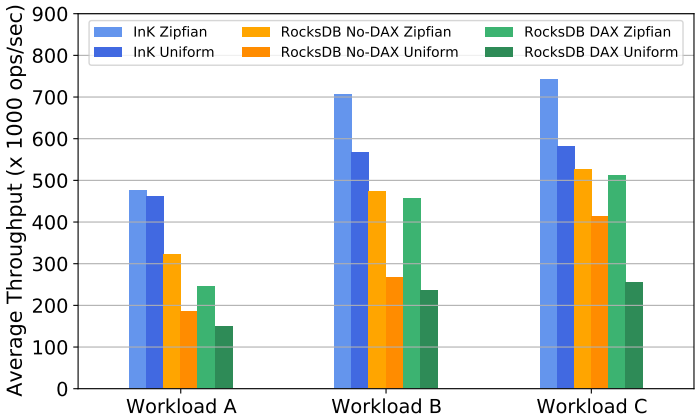
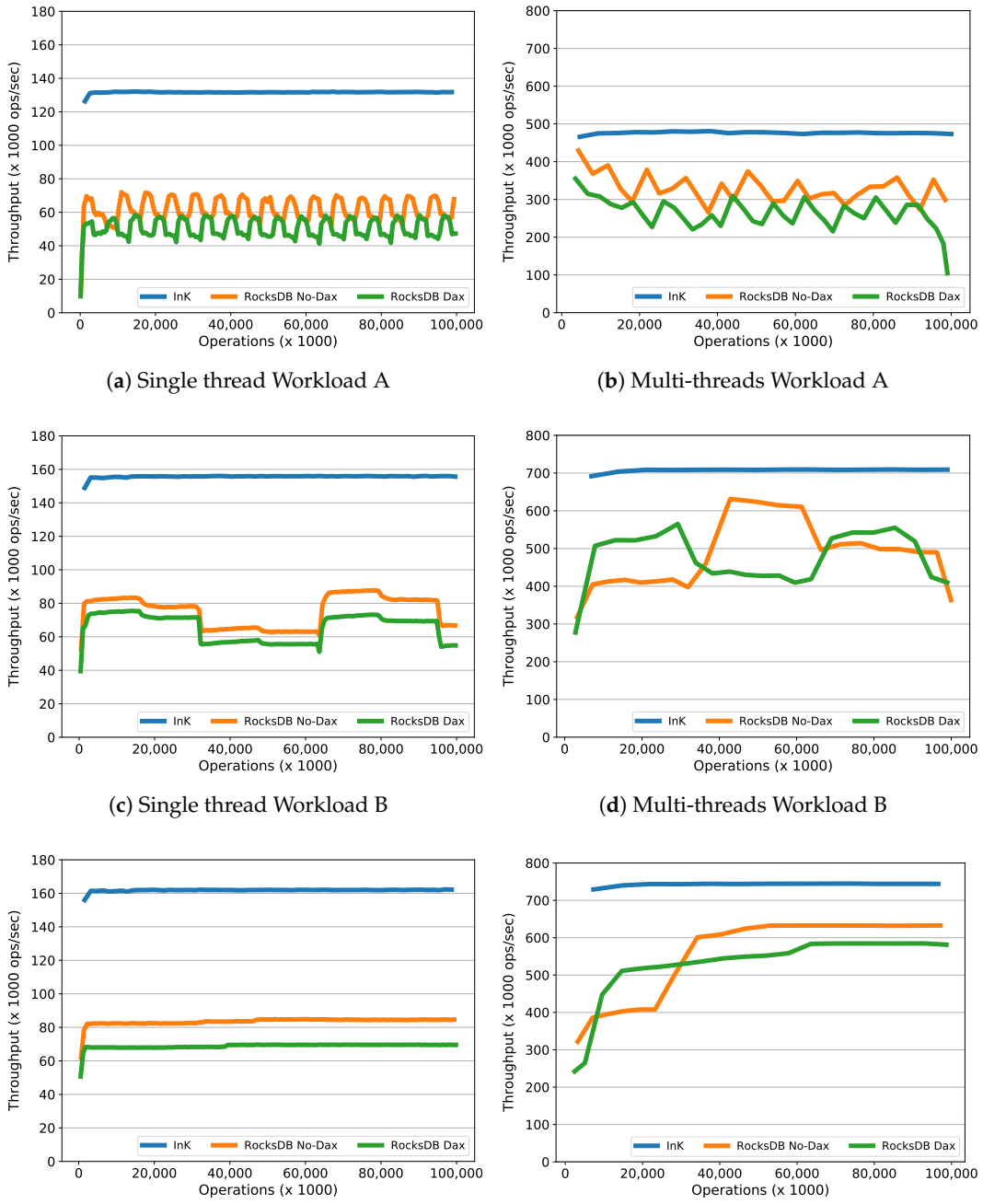


Figure 10. Performance comparison of InK and RocksDB.

We can see that InK outperforms RocksDB for all workloads and configurations. Specifically, InK can process 1.92, 1.54, and 1.44 times more throughput than RocksDB for workload A, B, and C, respectively. Both of InK and RocksDB show better performance when the accesses have a locality. The high locality gives a higher chance for the target key index to be cached. RocksDB has a higher chance to find the key from the higher level, shortening the lookup time.

Figure 11 plots the performance change over time while playing the YCSB workloads after initializing the system with 100 million key-value pairs. We also evaluate the performance implication of multithreads by comparing the performance of 1 thread to that of 10 threads. Over time, InK in blue line exhibits a consistent, stable performance whereas RocksDB in green and orange lines shows

significant performance fluctuation. The fluctuation is in a regular shape, and the more updates the workload has, the more often the performance fluctuates. We attribute the fluctuation to the memtable flush and compaction. RocksDB collects multiple key-value operations in memtable, and the flushes them to the storage. While collecting key-value pairs, RocksDB can exhibit a high performance, but when it starts flushing the memtable, the I/O path gets congested thereby reducing the performance. This reduced performance is restored when the flush is completed, and this fluctuation is repeated. Moreover, RocksDB may merge sstable on the storage through the process called compaction. This can incur a high I/O traffic, impairing the foreground performance. For this reason, RocksDB shows the continuous performance fluctuation.



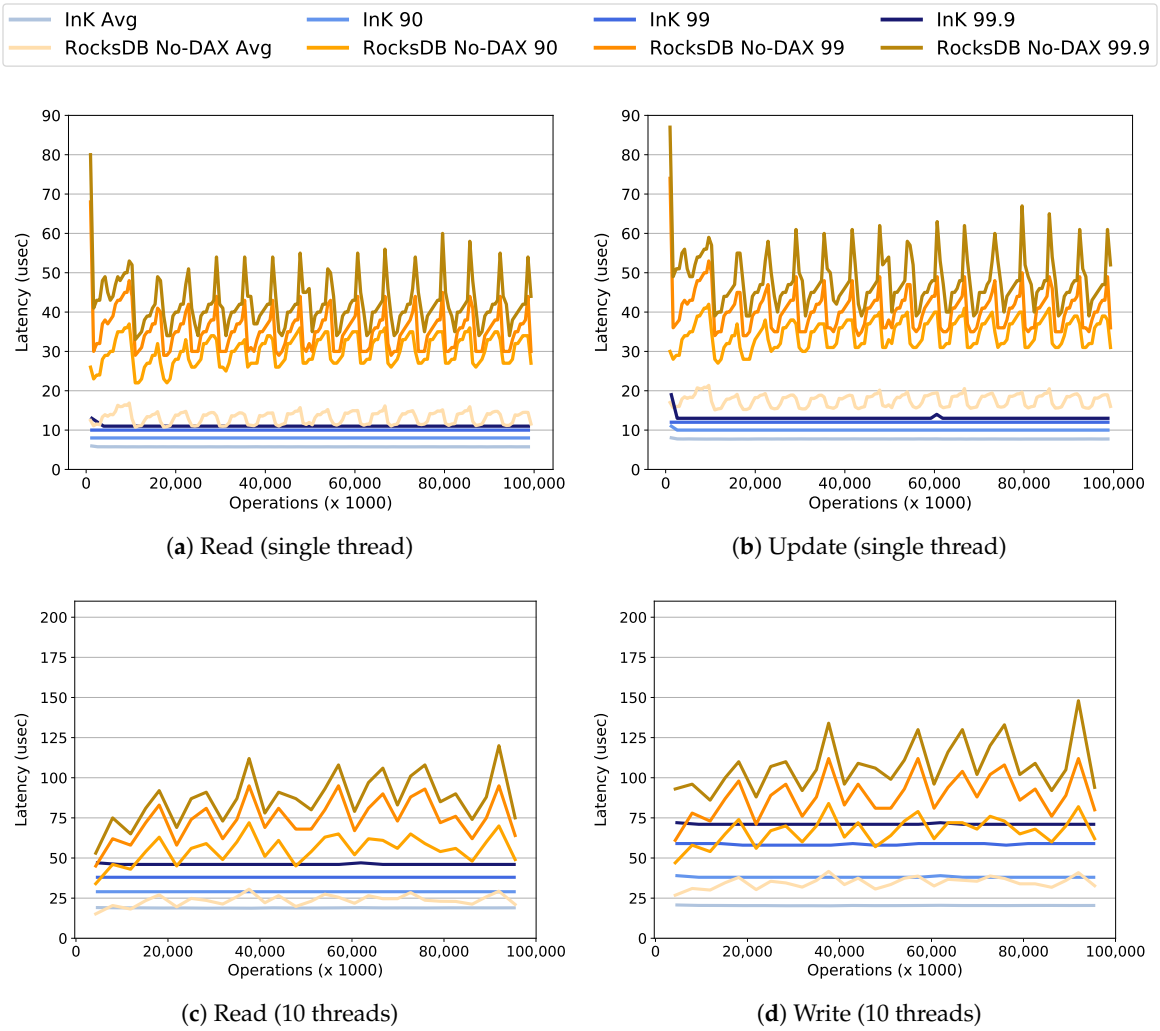
(**a**) Single thread Workload A (**b**) Multi-threads Workload A

(**c**) Single thread Workload B (**d**) Multi-threads Workload B

(**e**) Single thread Workload C (**f**) Multi-threads Workload C

Figure 11. Throughput comparisons of workloads with Zipfian distribution.

We further analyzed the performance variation of the key-value systems. Figure 12 shows the tail latency for YCSB Workload A. The chart shows the average, 90th percentile, 99th percentile, and 99.9th percentile of the latency. The rest of the conditions are identical to Figure 6.



(**a**) Read (single thread)

(**b**) Update (single thread)

(**c**) Read (10 threads)

(**d**) Write (10 threads)

Figure 12. Tail latency for YCSB Workload A. RocksDB is configured without DAX and key selection is in the Zipfian distribution.

InK shows very stable tail lattices in all configurations, which implies consistent performance of InK. Even in worst result, the most highest latency of InK is lower than the 90 percentile of RocksDB and lower than the average tail latency of RocksDB in the best case. It means that there is no big latency difference in each update and read operation since traversal time is identical for every index in leaf nodes. Thus, in both cases, the performance of InK has small variation. In contrast, in LSM tree, RocksDB may find the key from the memory in best case while it falls back to sstables in the worst case. It causes different traversing time between the operations depending on where the index we want is, making the large performance variation. This indicates that InK can provide a more stable performance, which is more preferred for key-value stores in practice.

5. Conclusions

In this paper, we propose InK, B+ tree-based in-kernel key-value storage for persistent memory. InK directly utilizes DCPM while other systems use file system to utilize DCPM. We implement InK in kernel, allowing applications to use InK through system calls. InK shows better update and read performance with guaranteeing the crash consistency and failure recovery. InK uses the lazy split

scheme for concurrent accesses and it makes several threads perform operations simultaneously. Theses approaches make InK as persistent memory aware design for a key-value storage. Evaluation with Yahoo Cloud Service Benchmark (YCSB) and RocksDB confirms that InK has advantages over LSM-tree based key-value store systems in terms of throughput and tail latency.

Author Contributions: Conceptualization, M.H. and S.-H.K.; methodology, M.H.; software, M.H.; validation, M.H. and S.-H.K.; formal analysis, M.H. and S.-H.K.; investigation, M.H. and S.-H.K.; resources, S.-H.K.; data curation, M.H.; writing–original draft preparation, M.H.; writing–review and editing, M.H. and S.-H.K; visualization, M.H.; supervision, S.-H.K.; project administration, S.-H.K.; funding acquisition, S.-H.K. All authors have read and agreed to the published version of the manuscript.

Funding: This research was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (2018R1C1B5085902) and Electronics and Telecommunications Research Institute (ETRI) grant funded by the Korean government (20ZS1310).

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Condit, J.; Nightingale, E.B.; Frost, C.; Ipek, E.; Lee, B.; Burger, D.; Coetzee, D. Better I/O Through Byte-Addressable, Persistent Memory. In Proceedings of the SOSP '09: ACM SIGOPS 22nd Symposium on Operating Systems Principles, Big Sky, MT, USA, 11–14 October 2009; pp. 133–146.
2. Volos, H.; Tack, A.J.; Swift, M.M. Mnemosyne: Lightweight Persistent Memory. In Proceedings of the ASPLOS XVI: 16th International Conference on Architectural Support for Programming Languages and Operating Systems, Newport Beach, CA, USA, 5–11 March 2011; pp. 91–104.
3. Apalkov, D.; Dieny, B.; Slaughter, J.M. Magnetoresistive Random Access Memory. *Proc. IEEE* **2016**, *104*, 1796–1830. [CrossRef]
4. Raoux, S.; Burr, G.W.; Breitwisch, M.J.; Rettner, C.T.; Chen, Y.; Shelby, R.M.; Salinga, M.; Krebs, D.; Chen, S.; Lung, H.; et al. Phase-change random access memory: A scalable technology. *IBM J. Res. Dev.* **2008**, *52*, 465–479. [CrossRef]
5. Apalkov, D.; Khvalkovskiy, A.; Watts, S.; Nikitin, V.; Tang, X.; Lottis, D.; Moon, K.; Luo, X.; Chen, E.; Ong, A.; et al. Spin-Transfer torque magnetic random access memory (STT-MRAM). *ACM J. Emerg. Technol. Comput. Syst.* **2013**, *9*, 1–35. [CrossRef]
6. Izraelevitz, J.; Yang, J.; Zhang, L.; Kim, J.; Liu, X.; Memaripour, A.; Soh, Y.J.; Wang, Z.; Xu, Y.; Dullor, S.R.; et al. Basic Performance Measurements of the Intel Optane DC Persistent Memory Module. *arXiv* **2019**, arXiv:1903.05714.
7. Lee, E.; Bahn, H.; Noh, S.H. Unioning of the Buffer Cache and Journaling Layers with Non-volatile Memory. In Proceedings of the FAST'13: 11th USENIX Conference on File and Stroage Technologies, San Jose, CA, USA, 12–15 February 2013; pp. 73–80.
8. Shin, W.; Chen, Q.; Oh, M.; Eom, H.; Yeom, H.Y. OS I/O Path Optimizations for Flash Solid-state Drives. In Proceedings of the ATC'14: 2014 USENIX Annual Technical Conference, Philadelphia, PA, USA, 19–20 June 2014; pp. 483–488.
9. Kim, H.J.; Lee, Y.S.; Kim, J.S. NVMeDirect: A User-space I/O Framework for Application-specific Optimization on NVMe SSDs. In Proceedings of the HotStorage'16: 8th USENIX Workshop on Hot Topics in Storage and File Systems, Denver, CO, USA, 20–21 June 2016.
10. Jeong, S.; Lee, K.; Lee, S.; Son, S.; Won, Y. I/O Stack Optimization for Smartphones. In Proceedings of the USENIX ATC'13: 2013 USENIX Conference on Annual Technical Conference, San Jose, CA, USA, 26–28 June 2013; pp. 309–320.
11. LevelDB: A Fast and Lightweight Key-Value Database. Available online: https://github.com/google/LevelDB (accessed on 10 May 2020).
12. Team, F.D.E. RocksDB: A Persistent Key-Value Store for Flash and RAM Storage. Available online: https://github.com/facebook/RocksDB (accessed on 12 March 2020).
13. Foundation, A.S. Apache Cassandra. Available online: https://github.com/apache/cassandra (accessed on 8 February 2020).
14. Foundation, A.S. Apache HBase. Available online: https://github.com/apache/hbase (accessed on 8 February 2020).

15. O'Neil, P.; Cheng, E.; Gawlick, D.; O'Neil, E. The log-structured merge-tree (LSM-tree). *Acta Inform.* **1996**, *33*, 351–385. [CrossRef]
16. Hwang, D.; Kim, W.H.; Won, Y.; Nam, B. Endurable Transient Inconsistency in Byte-addressable Persistent B+-tree. In Proceedings of the FAST'18: 16th USENIX Conference on File and Storage Technologies, Oakland, CA, USA, 12–15 February 2018; pp. 187–200.
17. Arulraj, J.; Levandoski, J.; Minhas, U.F.; Larson, P.A. Bztree: A High-Performance Latch-Free Range Index for Non-Volatile Memory. *Proc. VLDB Endow.* **2018**, *11*, 553–565. [CrossRef]
18. Kaiyrakhmet, O.; Lee, S.; Nam, B.; Noh, S.H.; Choi, Y.R. SLM-DB: Single-Level Key-Value Store with Persistent Memory. In Proceedings of the FAST'19: 17th USENIX Conference on File and Storage Technologies, Boston, MA, USA, 25–28 February 2019.
19. Liu, R.; Jin, P.; Wang, X.; Zhang, Z.; Wan, S.; Hua, B. NVLevel: A High Performance Key-Value Store for Non-Volatile Memory. In Proceedings of the 2019 IEEE 21st International Conference on High Performance Computing and Communications, Zhangjiajie, China, 10–12 August 2019.
20. Ginnaram, S.; Qiu, J.T.; Maikap, S. Role of the Hf/Si Interfacial Layer on the High Performance of MoS2-Based Conductive Bridge RAM for Artificial Synapse Application. *IEEE Electron Device Lett.* **2020**, *41*, 709–712. [CrossRef]
21. Maikap, S.; Banerjee, W. In Quest of Nonfilamentary Switching: A Synergistic Approach of Dual Nanostructure Engineering to Improve the Variability and Reliability of Resistive Random-Access-Memory Devices. *Adv. Electron. Mater.* **2020**, *6*, 2000209. [CrossRef]
22. Zarubin, M.; Kissinger, T.; Habich, D.; Willhalm, T.; Lehner, W. Efficient compute node-local replication mechanisms for NVRAM-centric data structures. In Proceedings of the DAMON'18: 14th International Workshop on Data Management on New Hardware, Houston, TX, USA, 11 June 2018; pp. 1–9.
23. Rudoff, A. Persistent Memory Programming. *Login USENIX Mag.* **2017**, *42*, 34–40.
24. Coburn, J.; Caulfield, A.M.; Akel, A.; Grupp, L.M.; Gupta, R.K.; Jhala, R.; Swanson, S. NV-Heaps: making persistent objects fast and safe with next-generation, non-volatile memories. In Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems, Newport Beach, CA, USA, 5–11 March 2011; pp. 105–118.
25. Dulloor, S.R.; Kumar, S.; Keshavamurthy, A.; Lantz, P.; Reddy, D.; Sankaran, R.; Jackson, J. System Software for Persistent Memory. In Proceedings of the EuroSys'14: 9th European Conference on Computer Systems, Amsterdam, The Netherlands, 13–16 April 2014.
26. Venkataraman, S.; Tolia, N.; Ranganathan, P.; Campbell, R.H. Consistent and durable data structures for non-volatile byte-addressable memory. In Proceedings of the FAST'11: 9th USENIX Conference on File and Stroage Technologies, San Jose, CA, USA, 15–17 February 2011; pp. 61–75.
27. Chen, S.; Jin, Q. Persistent B+-Trees in Non-Volatile Main Memory. *Proc. VLDB Endow.* **2015**, *8*, 786–797. [CrossRef]
28. Intel. Pmemkv. Available online: https://github.com/pmem/pmemkv (accessed on 25 October 2020).
29. Intel. PMDK: Persistent Memory Development Kit. Available online: https://pmem.io/pmdk (accessed on 25 October 2020).
30. Kannan, S.; Bhat, N.; Gavrilovska, A.; Arpaci-Dusseau, A.; Arpaci-Dusseau, R. Redesigning LSMs for Nonvolatile Memory with NoveLSM. In Proceedings of the USENIX ATC'18: 2018 USENIX Annual Technical Conference, Boston, MA, USA, 11–13 July 2018; pp. 993–1005.
31. Petrov, A. Algorithms behind Modern Storage Systems. *Commun. ACM* **2018**, *61*, 38–44. [CrossRef]
32. Wu, X.; Xu, Y.; Shao, Z.; Jiang, S. LSM-trie: An LSM-tree-based Ultra-Large Key-Value Store for Small Data Items. In Proceedings of the USENIC ATC'15: 2015 USENIX Annual Technical Conference, Santa Clara, CA, USA, 8–10 July 2015; pp. 71–82.
33. Yao, T.; Wan, J.; Huang, P.; He, X.; Gui, Q.; Wu, F.; Xie, C. A Light-weight Compaction Tree to Reduce I/O Amplification toward Efficient Key-Value Stores. In Proceedings of the MSST'17: 33rd International Conference on Massive Storage Systems and Technology, Santa Clara, CA, USA, 15–19 May 2017.
34. Cao, Z.; Dong, S.; Vemuri, S.; Du, D.H. Characterizing, Modeling, and Benchmarking RocksDB Key-Value Workloads at Facebook. In Proceedings of the FAST'20: 18th USENIX Conference on File and Storage Technologies, Santa Clara, CA, USA, 24–27 February 2020; pp. 208–223.
35. Atikoglu, B.; Xu, Y.; Frachtenberg, E.; Jiang, S.; Paleczny, M. Workload analysis of a large-scale key-value store. *SIGMETRICS Perform. Eval. Rev.* **2012**, *40*. [CrossRef]

36. Bhandari, K.; Chakrabarti, D.; Boehm, H.J. Implications of CPU Caching on Byte-Addressable Non-Volatile Memory Programming. Available online: https://www.hpl.hp.com/techreports/2012/HPL-2012-236.pdf (accessed on 11 March 2020)
37. Yang, J.; Kim, J.; Hoseinzadeh, M.; Izraelevitz, J.; Swanson, S. An Empirical Guide to the Behavior and Use of Scalable Persistent Memory. In Proceedings of the FAST'20: 18th USENIX Conference on File and Storage Technologies, Santa Clara, CA, USA, 24–27 February 2020; pp. 169–182.
38. Cooper, B.F. YCSB: Yahoo! Cloud Serving Benchmark. Available online: https://github.com/brianfrankcooper/YCSB (accessed on 7 June 2020).
39. Cooper, B.F.; Silberstein, A.; Tam, E.; Ramakrishnan, R.; Sears, R. Benchmarking Cloud Serving Systems with YCSB. In Proceedings of the SoCC'10: Symposium on Cloud Computing, Indianapolis, IN, USA, 10–11 June 2010; pp. 143–154.