
# HiKV: A Hybrid Index Key-Value Store for DRAM-NVM Memory Systems

**Fei Xia,** ***Institute of Computing Technology, Chinese Academy of Sciences; University of Chinese Academy of Sciences;*** **Dejun Jiang, Jin Xiong, and Ninghui Sun,** ***Institute of Computing Technology, Chinese Academy of Sciences***

https://www.usenix.org/conference/atc17/technical-sessions/presentation/xia

**This paper is included in the Proceedings of the 2017 USENIX Annual Technical Conference (USENIX ATC '17).**

**July 12–14, 2017 • Santa Clara, CA, USA**

ISBN 978-1-931971-38-6

**Open access to the Proceedings of the 2017 USENIX Annual Technical Conference is sponsored by USENIX.**

# HiKV: A Hybrid Index Key-Value Store for DRAM-NVM Memory Systems

Fei Xia[1,2], Dejun Jiang[1], Jin Xiong[1], and Ninghui Sun[1]

[1]SKL Computer Architecture, ICT, CAS [2]University of Chinese Academy of Sciences
{*xiafei2011,jiangdejun,xiongjin,snh*}*@ict.ac.cn*

## Abstract

Hybrid memory systems consisting of DRAM and Non-Volatile Memory are promising to persist data fast. The index design of existing key-value stores for hybrid memory fails to utilize its specific performance characteristics: fast writes in DRAM, slow writes in NVM, and similar reads in DRAM and NVM. This paper presents HiKV, a persistent key-value store with the central idea of constructing a hybrid index in hybrid memory. To support rich key-value operations efficiently, HiKV exploits the distinct merits of hash index and B$^+$-Tree index. HiKV builds and persists the hash index in NVM to retain its inherent ability of fast index searching. HiKV builds the B$^+$-Tree index in DRAM to support range scan and avoids long NVM writes for maintaining consistency of the two indexes. Furthermore, HiKV applies differential concurrency schemes to hybrid index and adopts ordered-write consistency to ensure crash consistency. For single-threaded performance, HiKV outperforms the state-of-the-art NVM-based key-value stores by reducing latency up to 86.6%, and for multi-threaded performance, HiKV increases the throughput by up to 6.4x under YCSB workloads.

## 1 Introduction

Emerging Non-Volatile Memory (NVM) technologies, such as PCM [1], ReRAM [2], and the recent 3D XPoint [3], are drawing substantial attentions from both academia and industry. One potential opportunity of NVM is to act as a fast persistent memory sitting on the memory bus, leading to hybrid DRAM-NVM memory systems [4, 5, 6]. Building storage systems, such as key-value stores, towards hybrid memory allows one to exploit fast memory access to achieve improved performance compared to basing on traditional hard disks or flash-based solid state drives (SSDs).

Persistent key-value stores (KV stores) have become an important part of storage infrastructure in data centers. They are widely deployed in large-scale production environments to serve search engine [7, 8], e-commerce platforms [9], social networking [10, 11], photo stores [12, 13], and more. In the past decade, there has been a large body of research on KV store design and optimization, on topics such as reducing write amplification to SSDs [14, 15, 16], reducing memory usage of indexing [17, 18, 19], and improving concurrency to achieve high scalability [11, 20, 21, 22]. Conventional KV stores are not suitable for hybrid memory systems because they are designed for the performance characteristics of hard disks or SSDs. For instance, many of existing studies adopt Log-Structured Merge Tree as the indexing structure [8, 11, 14, 15, 16], which avoids small random writes to hard disks or SSDs. Differing from hard disks and SSDs, hybrid memory systems are byte-addressable, and provide similar performance for sequential and random access. Maintaining sequential writes in large granularity instead introduces write amplification to NVM when designing KV stores for hybrid memory systems.

Indexing is a fundamental issue in designing key-value stores. The efficiency of supporting rich KV operations, such as *Put, Get, Update, Delete,* and *Scan*, is largely decided by the operational efficiency of indexing structure. For instance, searching B$^+$-Tree index is usually more costly than searching hash index. As we will show in Section 2.2, the operational efficiencies of different indexing structures are largely varied. Recently, a number of optimizations on B$^+$-Tree index are proposed for NVM memory systems [23, 24, 25, 26, 27, 28, 29, 30]. However, these techniques mainly focus on reducing consistency cost when directly persisting B$^+$-Tree index in NVM. On the other hand, the scalability of key-value stores is limited by the scalability of the indexing structure. For instance, partitioning the hash index allows one to scale the indexing structure to multiple threads, but partitioning the B$^+$-Tree index incurs expensive data movement when splitting large partitions or merging small ones. Thus, we argue that the choice of indexing structure for designing KV stores on hybrid memory is still open.

In this paper, we propose HiKV, a Hybrid index Key-Value store to run on hybrid memory. The central idea behind HiKV is the adoption of hybrid index: a hash index placed and persisted in NVM, and while a B$^+$-Tree index placed in volatile but fast DRAM without being persisted. The hybrid index fully exploits the distinct merits of the two indexes. It retains the inherent efficiency of hash operations to support single-key operations (*Get/Put/Update/Delete*). Moreover, it efficiently

accelerates *Scan* using the sorted indexing in B$^+$-Tree.

Adopting hybrid index introduces a number of challenges. First, when serving certain KV operations, including *Put*, *Update*, and *Delete*, the latency can be increased as HiKV needs to update two indexes to keep them consistent. HiKV solves this by placing the slow B$^+$-Tree index in fast DRAM and the fast hash index in slow NVM. In addition, HiKV updates the B$^+$-Tree index asynchronously to further hide its latency. Second, the scalability of the hybrid index requires careful design. Partitioning the hash index provides good scalability, while partitioning the B$^+$-Tree index suffers from high cost due to data migration. HiKV thus adopts partitioned hash indexes and a global B$^+$-Tree index. HiKV applies Hardware Transactional Memory (HTM) for the concurrency control of B$^+$-Tree index, and fine-grained locking to support concurrent accesses within individual hash index partitions. Finally, guaranteeing crash consistency of the hybrid index incurs expensive writes to NVM. HiKV adopts selective consistency that only ensures the consistency of hash index and key-value items by ordered-write. HiKV keeps the B$^+$-Tree index in DRAM and rebuilds it after system failure.

We implement HiKV and the state-of-the-art NVM-based key-value stores NVStore [28] and FPTree [30]. We evaluate the three KV stores using both micro-benchmarks and the widely used YCSB. For micro-benchmarks, HiKV can reduce latency by 54.5% to 83.2% and 28.3% to 86.6% compared with NVStore and FPTree, respectively. For YCSB workloads, HiKV outperforms NVStore by 1.7x to 5.3x, and FPTree by 24.2% to 6.4x in throughput.

This paper makes the following contributions:

1. We propose a hybrid index consisting of a hash index in NVM and a B$^+$-Tree index in DRAM to fully exploit the performance characteristics of hybrid memory to efficiently support rich KV operations.
2. We carefully design different concurrency schemes for the hybrid index to achieve high scalability with partitioned hash indexes and single global B$^+$-Tree index.
3. We propose ordered-write consistency and specific hash index design allowing atomic writes to ensure the crash consistency with reduced NVM writes.
4. We implement HiKV on top of the hybrid index. We conduct extensive evaluations to show the efficiency of the design choices of HiKV.

# 2 Background and Motivation

## 2.1 Non-Volatile Memory

Emerging Non-Volatile Memory (NVM) technologies, such as Phase Change Memory (PCM) [1] and Resistive Memory (ReRAM) [2], can provide faster persistence than traditional Disk and Flash. Table 1 shows the performance characteristics of different memory technologies. NVM provides similar read latency to DRAM, while its write latency is apparently longer than DRAM. Similar to NAND Flash, the write endurance of NVM is limited, especially for PCM. Thus, reducing writes to NVM is critical for software system design. At last, NVM has high performance for random accessing like DRAM, which is different from traditional Flash.

Table 1: Characteristics comparison of different memory technologies [28, 31, 32, 33, 34]

| Category | Read latency | Write latency | Write Endurance | Random accessing |
|---|---|---|---|---|
| DRAM | 60ns | 60ns | $10^{16}$ | High |
| PCM | 50∼70ns | 150∼1000ns | $10^{9}$ | High |
| ReRAM | 25ns | 500ns | $10^{12}$ | High |
| NAND Flash | 35us | 350us | $10^{5}$ | Low |

## 2.2 KV operations and indexing efficiency

The *Put*, *Get*, *Update*, and *Delete* are basic operations for KV stores. Besides, the *Scan* (short name for *Range Scan*) becomes important as required by today's applications. For instance, Facebook has replaced the storage engine of MySQL with a KV store MyRocks [35]. *Scan* turns out to be an important operation to serve range query of MySQL. Local file systems (i.e. TableFS [36]) and distributed file systems (i.e. CephFS [37]), use KV stores to store metadata. *Scan* is the core operation to support the second most prevalent metadata operation *readdir* [38]. Thus, efficiently supporting rich KV operations is significant for building key-value stores.

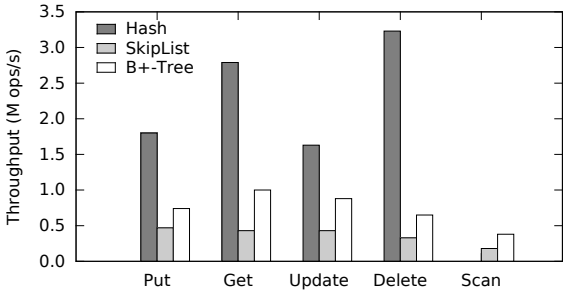



Figure 1: Throughput of different indexes

However, neither the hash indexing nor the sorted indexing can efficiently support all these operations. We use micro-benchmarks to quantify the efficiency of three widely used in-memory indexes: hash, skiplist, and B$^+$-Tree, to support the five KV operations. Figure 1 shows the in-memory throughput results with 50M key-values. For *Put/Get/Update/Delete*, hash index performs the most efficiently compared to the other two in-

dexes. Hash index usually involves less memory operations than skiplist and B$^+$-Tree, which requires multiple levels searching. However, as a non-sorted indexing, hash index provides extremely low throughput for *Scan* due to the cost of scanning the whole index space. Current NVM-based KV stores follow the widely adoption of B$^+$-Tree as the indexing structure. However, the above results motivate us to propose hybrid index to exploit distinguished merits of different indexes.

# 3 HiKV Design and Implementation

In this section, we present the system design and implementation of HiKV. We first present the design of the hybrid index. We then describe design issues when adopting hybrid index, including index updating, concurrency control, and crash consistency guaranteeing. Following that, we present the recovery of HiKV. At last, we describe the implementation of HiKV.

## 3.1 Hybrid index

Basic key-value operations include *Put, Get, Update, Delete,* and *Scan*[1]. To locate the requested key-value item, the single-key operations (*Put/Get/Update/Delete*) first take exactly one key to search the index. Once the KV item is located, *Get* directly returns the data, and while the write operations (*Put/Update/Delete*) require to persist updated index entry and new KV item if provided. Thus, the efficiency of index searching and data persisting is significant for these operations. Hash indexing inherently supports highly-efficient searching. Besides, regarding NVM reads perform similarly as in DRAM, placing a hash index in NVM as part of the hybrid index is a reasonable design choice. This design not only retains fast searching of hash index, but also allows persisting index in NVM directly without extra data copy from DRAM to NVM.

On the other hand, *Scan* takes a start key and count (or a start key and an end key) as input, which can benefit from sorted indexing. To efficiently support *Scan*, the hybrid index employs the widely used B$^+$-Tree index in main-memory systems [39, 40]. To maintain a consistent hybrid index, updating both hash index and B$^+$-Tree index for KV writes is fundamentally required. Updating B$^+$-Tree index involves many writes due to sorting as well as splitting/merging of leaf nodes. We thus place the B$^+$-Tree index in fast DRAM to avoid slow NVM writes in hybrid memory.

Figure 2 shows the architecture of hybrid index in hybrid memory. We discuss the issue of hybrid index updating in Section 3.2. Furthermore, to serve concurrent requests, the hash index is divided into multiple partitions. The B$^+$-Tree index is instead designed as a global one to indexes all KV items. We explain the differential concurrency schemes in Section 3.3. HiKV only guarantees the consistency of KV items and the hash index for improved performance. We present the ordered-write consistency mechanism in Section 3.4.

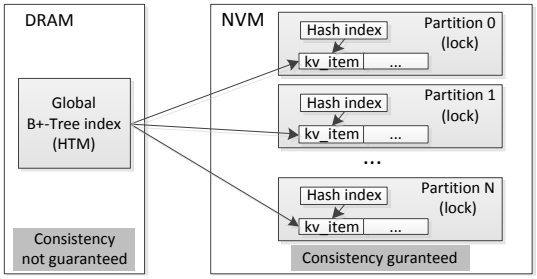



Figure 2: Architecture of hybrid index

## 3.2 Index updating

### 3.2.1 Asynchronous updates

When serving KV writes (*Put/Update/Delete*), HiKV needs to update both the hash index and the B$^+$-Tree index to keep them in a consistent state. One intuitive solution is to synchronously update both indexes. Due to the costly tree structure-specific operations, such as searching, sorting, splitting and merging, synchronous updates for the B$^+$-Tree index add extra latency to KV writes. Thus, HiKV employs asynchronous updates for hybrid index. In other words, HiKV retains synchronous updates to KV items and the hash index in NVM. For the B$^+$-Tree index in DRAM, HiKV asynchronously updates it in the background to hide the extra latency.

Figure 3 shows the procedure of HiKV to serve different KV operations. Taking *Put* as an example. HiKV first uses a *serving thread* to serve the incoming request. The serving thread is responsible for writing KV items to NVM (step1), and then writing the newly-added index entry to the hash index (step2). At last, the serving thread inserts the *Put* request to an updating queue (step3) and then returns. An asynchronous thread (called *backend thread*) gets requests from the updating queue and operates the B$^+$-Tree index in the background. In case of failing to update the B$^+$-Tree index due to system crash, HiKV can recover the B$^+$-Tree index from the hash index as presented in Section 3.5. In such doing, the observed latency of KV writes can be reduced.

However, a *Scan* request faces an inconsistent state of the B$^+$-Tree index as long as there exists requests in the updating queue when it arrives. Directly serving the scan request would retrieve old or invalid data. HiKV solves this by temporally blocking subsequent writes to enter into the updating queue once a scan is received. The

[1]Existing KV stores (i.e. Redis) support batch operations, such as MutiPut and MultiGet. HiKV can be extended to support such batch operations, which we leave as our future work.

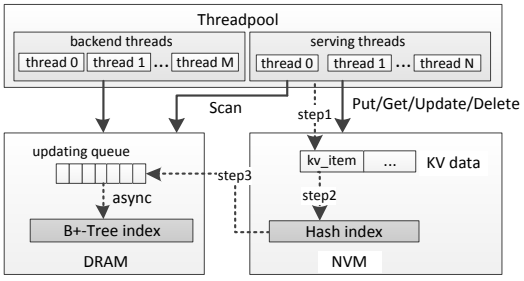



Figure 3: Procedure to serve KV operations

scan and subsequent writes wait until when all existing requests in the updating queue are processed. Once the updating queue becomes empty, it starts to receive further requests, and meanwhile the scan is served. Then, the concurrency control among the scan and subsequent writes on the B$^+$-Tree index are provided by Hardware Transactional Memory (HTM). We limit the length of the updating queue (i.e., holding 4096 requests in this paper) to avoid excessive waiting.

### 3.2.2 Dynamic threads adaption

To serve highly-concurrent requests, HiKV needs to increase the number of serving threads. For read-write mixed workloads, this can rapidly fill the updating queue as many serving threads put write requests into the queue. If the backend threads fall behind the serving threads, the updating queue becomes full and further blocks serving subsequent requests. Thus, HiKV needs to dynamically adapt the backend threads according to the change of serving threads.

We usually set a fix-sized thread pool to run both serving threads and backend threads. The dynamic threads adaption scheme is to decide the numbers of serving threads ($N_{sthd}$) and backend threads ($N_{bthd}$). Basically, we need to match the average processing rate of the backend threads on the updating queue with the average queue filling rate of the serving threads. The processing rate and filling rate are determined by a number of factors, such as KV operation complexity, ratio of different KV operations, and DRAM/NVM performance. To decide $N_{sthd}$ and $N_{bthd}$ at runtime, we sample the numbers of different KV operations as well as their average operational latencies. Based on our observation, the operational latency of *Scan* is 14 times than that of *Get*, and the latency gap among *Put*, *Update*, and *Delete* is less than 2x. For simplicity, we do not distinguish *Put/Update/Delete* when sampling but sample *Scan* and *Get* operations separately. Within each sampling window, assuming the number of writes is normalized to 1, the number of *Get* is $N_g$ and the number of *Scan* is $N_s$. The average latencies of *Get* and *Scan* are $L_g$ and $L_s$. The average write latencies of backend thread and serving thread are $L_{bw}$ and $L_{sw}$, respectively. Then, $N_{sthd}$ and $N_{bthd}$ should satisfy the following two equations, in which $N_{thd}$ is the total size of the thread pool. In such doing, the average processing latency of backend threads matches the one of serving threads.

$$(N_g \cdot L_g + N_s \cdot L_s + 1 \cdot L_{sw})/N_{sthd} = (1 \cdot L_{bw})/N_{bthd} \quad (1)$$

$$N_{sthd} + N_{bthd} = N_{thd} \quad (2)$$

## 3.3 Differential concurrency

Concurrency control is a key issue for improving the scalability of KV stores in the multi-core era. In this section, we present the differential concurrency schemes applied to the hybrid index.

Partitioning is shown to achieve high throughput and scalability for balanced workloads [40]. Thus, HiKV adopts the widely-used keyhash-based partitioning [41, 37, 42] for the hash index. All KV items are distributed to multiple partitions according to the hash value of the key, and each partition uses a hash index as Figure 2 shows. HiKV allows concurrent accessing to a partition by multiple threads to handle skewed workloads. It uses fine-grained locking for concurrency control inside each partition. HiKV uses an atomic write to update the hash index entry as illustrated in Section 3.4. As a result, HiKV can read an index entry when another thread is updating it without locking.

Partitioning the B$^+$-Tree index results in either *unordered multi-B$^+$-Tree indexes* as in Cassandra [43] and Megastore[44] using keyhash-based approach, or *ordered multi-B$^+$-Tree indexes* as in SLIK [45] using range partition. However, we argue that none can efficiently support *Scan* due to extra efforts. With unordered multi-B$^+$-Tree indexes, we need to issue the scan request to all indexes, and then return the matching key-values from the result. Such approach increases the concurrency overhead. With ordered multi-B$^+$-Tree indexes, the scan request can be only issued to indexes that contain corresponding KV items. However, such approach needs to migrate index entries when an index becomes too large or too small, which degrades system performance. Thus, HiKV adopts a global B$^+$-Tree index for all KV items in NVM. HiKV adopts HTM to handle concurrency control of the global B$^+$-Tree index.

## 3.4 Ordered-write consistency

Guaranteeing crash consistency is a fundamental requirement for persistent KV stores. Since NVM has long write latency, HiKV needs to reduce NVM writes when guaranteeing consistency. We first apply selective consistency to HiKV to only ensure the consistency of hash index and KV items, but not guarantee consistency for the B$^+$-Tree index to avoid expensive data copy from

DRAM to NVM. Upon a system failure, HiKV recovers the B$^+$-Tree index as presented in Section 3.5.

Secondly, we apply ordered-write to ensure the consistency of the hash index and KV items. Conventional logging and copy-on-write incur two writes when guaranteeing consistency. The ordered-write consistency first updates the KV item out-of-place. Then, it updates the hash index in-place using an atomic write. A KV item is not visible until the atomic write is finished. In such doing, crash consistency is guaranteed without introducing extra writes. We then describe the specific hash index design for supporting atomic write and present the consistency algorithms.

### 3.4.1 Hash index design

Modern processors support 8B atomic writes natively and 16B atomic writes using cmpxchg16b instruction (with LOCK prefix) [46, 47]. However, the key size of KV stores is usually 16B [13, 48]. Directly placing the original key and the position of KV item in a hash index entry makes it impossible to apply atomic writes.

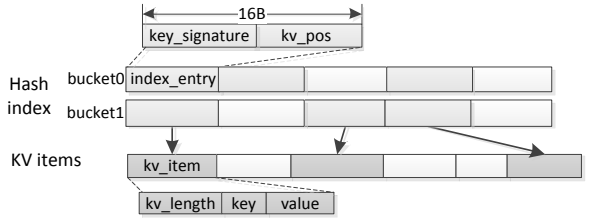



Figure 4: Hash index and key-value data layout

The position of key-values needs 48bits in modern processors. If the index entry is designed to be 8B, then the key signature can only occupy 16 bits. There exists many signature conflicts as 16 bits signature can only distinguish 65536 keys. Thus, HiKV adopts 16B index entry that can also be updated atomically. Figure 4 shows the design of the hash index and KV items. A 16B index entry consists of a 64bits key signature and a 64bits position to refer the position of KV item. A hash bucket contains multiple 16B index entries. To support varied-length key and value, each KV item stores a 32bit kv_length, key and value. Key signature may still be conflicted among different keys. Thus, HiKV checks corresponding KV item if the key signature in index entry equals to the signature of specified key.

### 3.4.2 Consistency algorithm

In this subsection, we present the consistency algorithms of different HiKV operations. Note that, memory writes may be reordered due to the caching of CPU or the scheduling of memory controller. HiKV uses the sequence of *sfence, clflush, sfence* instruction (referred to *persist*) to enforce the ordering and persistency of memory writes based on existing hardware [24, 27, 28, 49, 50]. The *clflush* can be replaced with the latest *CLWB* instruction [51] if the hardware supports it.

**Put.** Algorithm 1 presents the pseudo-code of *Put*. It first finds an empty index entry (line 1). Then the algorithm allocates free space to store the KV item (line 2). Next, it sets the KV item (line 3), and persists the KV item to NVM (line 4). At last, it performs an atomic write to set the index entry(line 5), and persists the index entry (line 6).

**Algorithm 1** HiKV_PUT(op, key, value)

```
1: index_entry = find_empty_entry(key);
2: new_kv_item = alloc_space(sizeof(kv_item));
3: set new_kv_item according to key and value;
4: persist(new_kv_item, sizeof(kv_item));
5: AtomicWrite(index_entry, new_entry);
6: persist(index_entry, sizeof(index_entry));
```

**Update.** Algorithm 2 presents the pseudo-code of *Update*. The algorithm finds the original index entry according to the key (line 1), and uses the index entry to find the original KV item in NVM (line 2). Since HiKV adopts out-of-place update for KV item, it needs to allocate free space to store new KV item (line 3). Then, it sets the KV item, persists it, atomically updates the index entry, and persists it like *Put* (line 4-7). At last, it deallocates the space of original KV item (line 8).

**Algorithm 2** HiKV_UPDATE(op, key, value)

```
1: index_entry = find_index_entry(key);
2: orig_kv_item = get_original_item(index_entry);
3: new_kv_item = alloc_space(sizeof(kv_item));
4: set new_kv_item according to key and value;
5: persist(new_kv_item, sizeof(kv_item));
6: AtomicWrite(index_entry, new_entry);
7: persist(index_entry, sizeof(index_entry));
8: free_space(orig_kv_item);
```

**Delete.** Algorithm 3 presents the pseudo-code of the *Delete* operation. The algorithm first finds the original index entry and KV item (line 1, 2). It invalids the index entry by setting it to 0 using an atomic write (line 3), and then persists the index entry (line 4). At last, it deallocates the space of original KV item (line 5).

**Algorithm 3** HiKV_DELETE(op, key)

```
1: index_entry = find_index_entry(key);
2: orig_kv_item = get_original_item(index_entry);
3: AtomicWrite(index_entry, 0);
4: persist(index_entry, sizeof(index_entry));
5: free_space(orig_kv_item);
```

The validity of a KV item is identified by corresponding index entry. Since the index entry is atomically updated at last, crashes happened in any step of the three algorithms do not destroy consistency.

Note that, HiKV faces the challenge of memory leak when a crash occurs after allocating a free NVM space. Solving memory leak thoroughly relies on the support of underlying libraries and operating system. We leave it as our future work.

## 3.5 Recovery

In this section, we describe the recovery of HiKV after normal shutdown and system failure.

**Recovery after a normal shutdown.** On a normal shutdown, HiKV persists the $B^+$-Tree index in continuous NVM space. Then, HiKV saves the start address of this space to a reserved position in NVM and atomically writes a flag indicating a normal shutdown. HiKV checks the flag when it recovers the index. If the flag indicates a normal shutdown, then HiKV reads the $B^+$-Tree index stored in NVM and recovers it to DRAM. Otherwise, HiKV executes the following recovery.

**Recovery after a system failure.** In case of a system failure, HiKV must rebuild the $B^+$-Tree index from the consistent hash index and key-value items in NVM by only scanning all hash indexes. For each index_entry in every hash_index, if its value is not zero, the recovery thread inserts the key and the position of corresponding KV item to the $B^+$-Tree index. Otherwise, the index_entry is invalid.

## 3.6 Implementation

We implement HiKV on top of the hybrid index. HiKV utilizes the lossless hash index design in MICA [42]. A hash bucket contains multiple successive index entries. HiKV sequentially searches next index entry in the hash bucket when a hash collision occurs. Each index entry in the leaf nodes of $B^+$-Tree contains a whole key and the position of corresponding KV item in NVM. We implement multiple lock-free updating queues to reduce contention when serving highly concurrent requests. All backend threads poll updating queues as the cost of thread synchronization is high.

# 4 Evaluation

In this section, we evaluate the performance of HiKV. We first describe the experimental setup and then evaluate HiKV using micro- and macro-benchmarks.

## 4.1 Experimental Setup

We conduct all experiments on a server equipped with two Intel Xeon E5-2620 v4 processors. Each one running at 2.1 GHz has 8 cores, a shared 20MB last level cache. The memory size in the server is 256GB.

**NVM emulation**

As real NVM DIMMs are not available for us yet, we emulate NVM using the DRAM similar to prior works [49, 52, 6]. The access latency of DRAM is about 60 ns [49], and the write latency of the latest 3D-XPoint is ten times of DRAM [3]. Thus, we set the NVM write latency to 600 ns. We add extra write latency only once for each *persist* operation as described in Section 3.4.2. We add the long write latency of NVM using the x86 RDTSCP instruction. We use the instruction to read the processor timestamp counter and spin until the counter reaches the configured latency. We do not add extra read latency for NVM as it has similar read latency with DRAM [28, 31]. The impact of longer NVM read latency is evaluated in Subsection 4.6.

**Workloads**

We use five micro-benchmarks to evaluate the performance of single KV operations, namely *Put*, *Get*, *Update*, *Delete*, and *Scan*. The randomly generated scan count is less than 100 like YCSB [53]. For each micro-benchmark, we first warm up KV stores with 50M key-values. Then, we execute 50M operations with randomly selected key-values. All our micro-benchmarks generate KV data following the uniform distribution. We use the widely used macro-benchmark YCSB to evaluate the performance of mixed operations. We also execute 50M key-value operations. We use the default configuration of YCSB that is zipfian distribution with 99% skewness.

For both micro- and macro- benchmarks, we always use a key size of 16B, which is a typical size in production environment [13, 48]. In Facebook, over 90% value sizes of Memcached are close but less than 500B [48]. Thus, we set the value size to 256B basically.

**Compared systems**

We compare HiKV with the state-of-the-art NVM KV store NVStore [28] and hybrid memory KV store FPTree [30]. We do not compare HiKV with disk-based KV stores, such as RocksDB [11]. This is because HiKV is designed for byte-addressable NVM, and its I/O stack is quite different from that of RocksDB. We also do not evaluate KV stores that periodically persist data, such as Echo [54] and Masstree [39]. These KV stores cannot guarantee the consistency of every KV operation.

We re-implement NVStore and FPTree as faithfully as possible according to the descriptions in their papers. The index of NVStore is an optimized $B^+$-Tree, called NVTree, which keeps entries in leaf nodes unsorted to reduce NVM writes. To be fair, we place inner nodes of NVTree in the DRAM as the way HiKV uses the DRAM. FPTree also uses a variation of $B^+$-Tree, which adds a bitmap and fingerprints in each unsorted leaf node to accelerate searching.

A typical usage of DRAM for hybrid memory systems is using DRAM as a cache of NVM, besides placing a

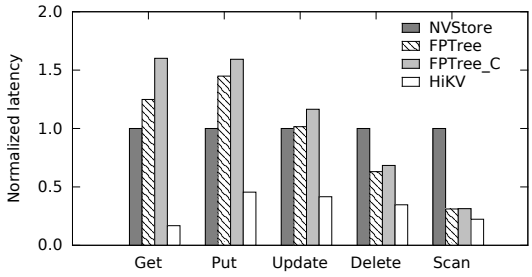



Figure 5: Normalized latency of KV ops

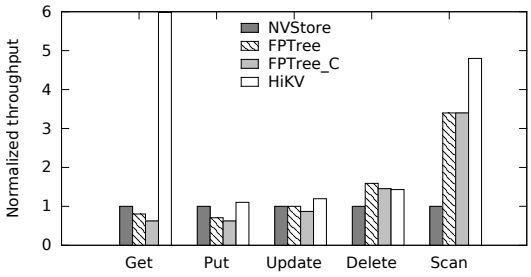



Figure 6: Normalized throughput of KV ops

part of index in DRAM. For 16B keys and 256B values, HiKV's ratio of DRAM consumption to NVM is 15.4% larger than FPTree (details in Subsection 4.7). Thus, we use the extra DRAM as a cache of FPTree, called FPTree_C in our evaluation. FPTree_C uses hash index and LRU replacement policy to manage the cache.

## 4.2 Single-threaded performance

We first evaluate the single-threaded performance of HiKV using micro-benchmarks. For benchmarks that only read data, including *Get* and *Scan*, all four KV stores use one thread. Note that, HiKV is designed to adopt serving threads accompanied with backend threads to operate the $B^+$-Tree index when serving write requests. Thus, for *Put*, *Update*, and *Delete*, HiKV is configured to use one serving thread and one backend thread. For fair comparison, both NVStore, FPTree, and FPTree_C are configured with two threads.

### 4.2.1 Latency reduction

Figure 5 shows the latency reduction of HiKV. For *Get*, HiKV can reduce latency by 83.2% and 86.6% than NVStore and FPTree, respectively. HiKV only needs to lookup the fast hash index. However, both NVStore and FPTree not only need to lookup the tree index, but also need to sequentially lookup a leaf node as keys in the leaf node are unsorted. For *Put*/*Update*/*Delete*, HiKV can reduce latency by 54.5%/58.4%/65.3% than NVStore, and 68.8%/59.1%/45.0% than FPTree, respectively. This is because searching the hash index is fast and HiKV uses asynchronous mechanism to hide the latency of $B^+$-Tree index. For *Put* and *Update*, FPTree needs to persist data three times (bitmap, fingerprints and key-value), while NVStore only needs to persist data twice (key-value and leaf.number). As a result, the *Put* and *Update* latencies of FPTree are higher than those of NVStore. For *Delete*, HiKV only needs to invalid the corresponding index_entry and persist it to NVM. However, NVStore needs to insert the key-value with an invalid flag and update the leaf_number, which persists data to NVM twice. Although FPTree only needs to invalid bitmap and persist once for *Delete*, its latency is still larger than that of HiKV due to inefficient searching of tree index.

For *Scan*, HiKV can reduce latency by 77.7% and 28.3% than NVStore and FPTree, respectively. When putting a key-value, NVStore does not check whether the key-value has existed in the leaf node. As a result, it must check whether a key-value is valid or not when scanning a leaf node. Thus, the *Scan* latency of NVStore is apparently larger than that of HiKV. FPTree uses a bitmap per leaf node to identify the validity of key-value entries in the leaf node. Thus, the *Scan* latency of FPTree is lower than that of NVStore, while it is still larger than that of HiKV.

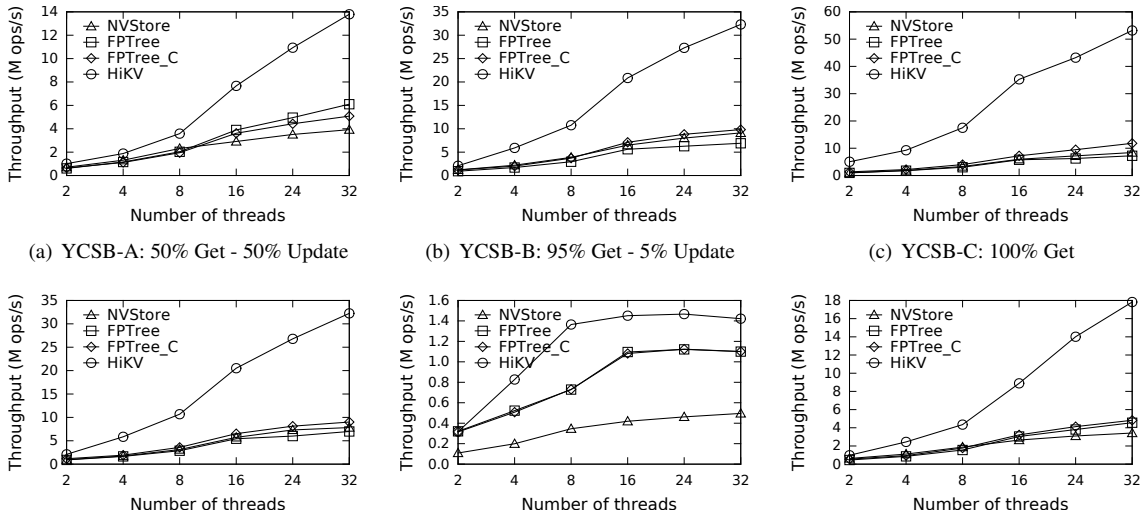
FPTree_C performs worse than FPTree for single-key operations. This is because the micro-benchmarks have uniform distribution, which results in low cache hit ratio. FPTree_C incurs extra performance overhead for the cache replacement.

### 4.2.2 Throughput improvement

Figure 6 shows the throughput improvement of HiKV. HiKV can improve throughput by 5.0x/3.8x than NVStore, and 6.4x/41.2% than FPTree for *Get*/*Scan*, respectively. For *Put*/*Update*, HiKV outperforms NVStore and FPTree by 10.4%/19.6%, and 55.9%/19.6%, respectively. The *Delete* throughput of HiKV is 43.2% higher than that of NVStore, and 10.0% lower than that of FPTree. The throughput improvement of HiKV is lower in *Put*/*Update*/*Delete* than in *Get*/*Scan*. This is because NVStore and FPTree use two threads to run these write requests, while HiKV only uses one serving thread. For read requests, these three KV stores use one thread. FPTree_C achieves lower throughput than FPTree due to the overhead of DRAM cache management.

## 4.3 Scalability

We then evaluate the scalability of HiKV using the macro-benchmark YCSB. We do not use the original YCSB framework with client-server mode due to its long latency of network stack. Here, we use a local YCSB workload generator following the default YCSB configurations like MICA [42]. HiKV dynamically adapts the number of serving threads and back-




(a) YCSB-A: 50% Get - 50% Update

(b) YCSB-B: 95% Get - 5% Update

(c) YCSB-C: 100% Get

(d) YCSB-D: 95% Get - 5% Put

(e) YCSB-E: 95% Scan - 5% Put

(f) YCSB-F: 50% Get - 50% ReadModify-Write

Figure 7: Scalability on YCSB workloads

end threads according to different workloads. To be fair, we configure NVStore, FPTree, and FPTree_C to use the same total number of threads as HiKV in all evaluations. We run these benchmarks at most 32 threads as the server we used has 16 cores.

Figure 7 presents the throughputs of YCSB workloads as the number of threads varies from 2 to 32. The performance of HiKV with 32 threads is increased by a factor of 13.6/15.5/10.5/15.4/4.3/18.3 for YCSB-A/B/C/D/E/F against the two-threaded execution, respectively. For the same scalability evaluation, the scaling factors for NVStore, FPTree, and FPTree_C are 5.5/7.6/8.2/8.2/4.6/5.5, and 10.0/7.5/7.5/7.7/3.4/10.1, and 7.9/8.2/8.8/7.8/3.5/8.8, respectively. In summary, HiKV achieves better scalability than NVStore and FPTree. The *Get* ratio is 95%, 95%, and 75% for YCSB-B, YCSB-D, and YCSB-F, respectively. HiKV provides more than 20 serving threads with 32-threaded execution due to the dynamic threads adaption. However, HiKV only has one serving thread with 2-threaded execution. As a result, HiKV executed with 32 threads can improve throughput by large than 15x than 2-threaded execution for YCSB-B/D/F.

With 32-threaded execution, HiKV outperforms NVStore by 1.7x to 5.3x, FPTree by 24.2% to 6.3x, and FPTree_C by 24.1% to 3.5x. For read-intensive and skewed workloads, such as YCSB-B/C, FPTree_C performs better than FPTree for as the cache hit ratio is high. For YCSB-E, HiKV can scale to 8 threads almost linearly and keeps stable with more threads. This is because HiKV must lock all updating queues temporally before serving *Scan*, which would block the *Put* of other threads. NVStore, FPTree and FPTree_C can scale to 16 threads for YCSB-E. Even so, HiKV still improves throughput by 1.7x, 24.2%, and 24.1% than NVStore, FPTree and FPTree_C, respectively.

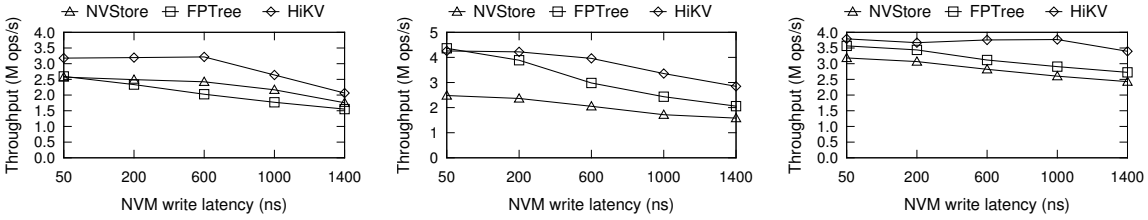
## 4.4 Sensitivity analysis

In this section, we conduct sensitivity analysis to HiKV considering NVM write latency and workload dataset size. We use 16 threads for all the experiments.

### 4.4.1 Sensitivity to NVM write latency

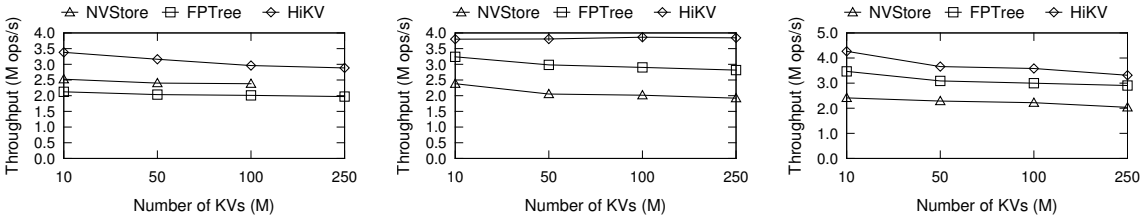
The write latencies are different among various NVM devices. Thus, we evaluate the impact of NVM write latency on the performance. Figure 8 shows the throughput results when we vary NVM write latency from 50 ns to 1400 ns. The *Get* and *Scan* performance have no relation with the write latency. Thus, we only show the results of *Put*, *Update*, and *Delete*. We do not show the results of FPTree_C as it performs worse than FPTree for uniform distributed workloads.

We find that the throughput decreases as NVM write latency increases for NVStore and FPTree. This is due to the increase of persist latency. For *Delete*, the throughput of HiKV remains stable when the write latency is lower than 1400 ns. This is because the concurrent deletion latency of $B^+$-Tree index is still longer than that of the hash index even though the write latency increases to 1000 ns. Compared to NVStore and FPTree, HiKV still improves the throughput of *Delete* by 17.6%/80.0%/39.9%, and 32.9%/38.4%/24.6% for *Put*/*Update*/*Delete*, respectively even if the write latency of NVM reaches 1400 ns.




(a) Put (b) Update (c) Delete

Figure 8: Impact of NVM write latency




(a) Put (b) Update (c) Delete

Figure 9: Impact of dataset size

### 4.4.2 Sensitivity to dataset size

HiKV adopts a global B$^+$-Tree index to support *Scan*. A doubt is whether the B$^+$-Tree index limits the scalability of HiKV as the dataset size increases. Figure 9 shows the throughput as the number of key-values increases from 10M to 250M. The *Put* throughput of NVStore is not available for 250M key-values as they run out of our server memory due to the sharply increased size of the NVTree index. The total number of key-values is 500M for *Put* as we first warm up with 250M key-values.

The throughput of HiKV remains unchanged for *Update*, while the throughput of NVStore and FPTree decreases by 19.3% and 13.1%, respectively. When the number of key-values increases 25 times, the throughput of HiKV, NVStore, and FPTree decreases by 14.6% /22.4%, NA/15.6%, and 7.2%/16.3% for *Put* and *Delete*, respectively. The performance degradation is due to the increased searching latency with increased dataset size. The update throughput of HiKV is determined by serving threads under such configuration. Thus, we can conclude that the global B$^+$-Tree index does not limit the scalability compared to NVStore and FPTree.

## 4.5 Performance breakdown

In this section, we first analyze the effectiveness of asynchronous updates, differential concurrency, and ordered-write consistency of HiKV. HiKV_sync updates the hash index and B$^+$-Tree synchronously within one thread. HiKV_par adopts partitioning-based concurrency control for B$^+$-Tree index, which has ordered multi-B$^+$-Tree indexes. HiKV_wal uses the traditional Write-Ahead Log to guarantee consistency.

Figure 10 shows the average latency of *Put* as the NVM write latency increases from 50 ns to 1400 ns. Compared with HiKV_sync, HiKV can reduce latency by 46.7% to 57.8%. This is due to the asynchronous updates of HiKV that the critical path only contains operating the hash index. HiKV can reduce latency by 11.2% to 17.4% compared to HiKV_par. The performance degradation of HiKV_par is caused by the two reasons. First, migrating index entries among B$^+$-Tree indexes blocks normal put operations. Second, the migration thread preempts CPU resources in 16-threaded execution. HiKV_wal stores key and value position (index_entry) in hash index. To guarantee consistency, HiKV_wal first writes key-value in log area, then write value to NVM and writes the index_entry in hash index for *Put*. Writing value and index_entry in hash table without logging will result in inconsistency. This is because the index_entry and value in HiKV_wal can not be update atomically. We can find that HiKV_wal needs to persist data to NVM three times although HiKV_wal does not need to guarantee the order of writing value and index_entry to NVM. However, HiKV only needs to persist data to NVM twice due to the order-writing. The evaluation result shows that HiKV can reduce latency than HiKV_wal by up to 27.4% when the NVM write latency reaches 1400 ns.

Secondly, we evaluate the effectiveness of dynamic threads adaption in HiKV. We first warmup 50M key-values, then we execute back-to-back YCSB-

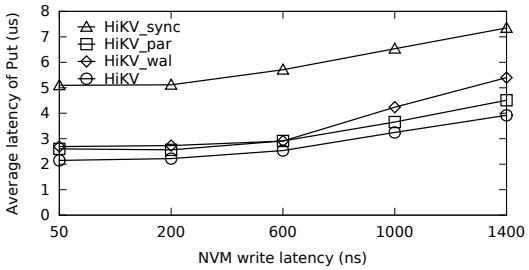



Figure 10: Effectiveness of asynchronous updates, differential concurrency, and order-writing consistency

A/B/C/D/E/F. The percentages of *Put/Get/Update/Scan* are varied in these workloads. Each workload is executed by 60 sec. The total number of threads is 16. HiKV-8-8 and HiKV-12-4 represents executing with static 8 serving threads and 8 backend threads, and 12 serving threads and 4 backend threads, respectively.

Figure 11 shows the throughputs as the workload changes from YCSB-A to YCSB-F. HiKV can achieve the highest throughput except for YCSB-A. For YCSB-B/C/D/E/F, HiKV outperforms HiKV-8-8 and HiKV-12-4 by 10.5% to 1.0x and 10.4% to 37.5%, respectively. For YCSB-A, the throughput of HiKV is same throughput with HiKV-8-8, and slightly lower than HiKV-12-4 by 1.6%. HiKV dynamically adapts the number of serving threads and backend threads, such as 8 and 8 for YCSB-A, 13 and 3 for YCSB-B, 9 and 7 for YCSB-F. For read-intensive workloads, increasing the number of serving threads can improve throughput of HiKV.

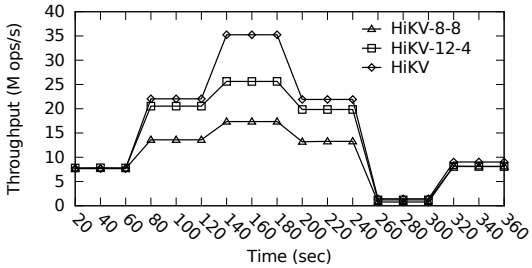



Figure 11: Effectiveness of dynamic threads adaption. *0-60s:YCSB-A. 60-120s:YCSB-B. 120-180s:YCSB-C. 180-240s:YCSB-D. 240-300s:YCSB-E. 300-360s:YCSB-F.*

## 4.6 Impact of NVM read latency

A few researches indicate that the read of NVM is longer than that of DRAM [47, 55]. Thus, we evaluate the impact of NVM read latency on system performance. We emulate the longer read latency similar to emulating write latency. We set the NVM read latency to 120 ns, which is twice as that of DRAM [55].

Figure 12 shows that the average serving latency increases as NVM read latency does. This is because HiKV spends more time to search hash index. However, NVStore and FPTree also takes more time when searching unsorted leaf nodes and splitting/merging leaf nodes. Thus, HiKV can still apparently reduce latency than NVStore and FPTree. For example, HiKV can reduce latency by 80.0%/61.8% than NVStore, and 82.3%/13.0% than FPTree for *Get*/*Scan* with doubled read latency, respectively.

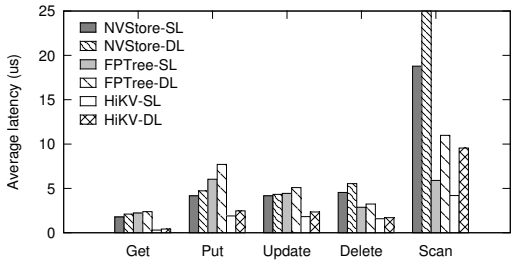



Figure 12: Impact of NVM read latency. (*SL* and *DL* represent same latency and doubled latency as DRAM.)

## 4.7 Memory consumption

Figure 13 shows the DRAM and NVM consumptions after randomly putting 50M key-values to different KV stores. The value size varies from 64B to 1KB. The curves show the ratio of DRAM consumption to NVM consumption. Since DRAM is used to store the indexes of key-values, the DRAM consumptions are related to the number of key-values and keep constant with varied value sizes for both KV stores. On the contrary, in both KV stores, NVM is used to store data items and its consumption increases as the value size increases.

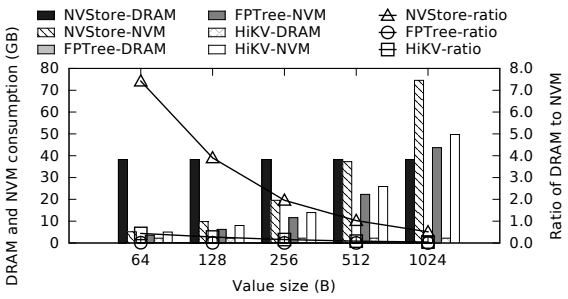



Figure 13: DRAM consumption

We observe that NVStore consumes DRAM as high as 38.27 GB. This is mainly because NVStore creates one parent-leaf-node per leaf node when rebuilding the contiguous inner nodes of NVTree. The index size of NVTree increases exponentially as the tree height increases. HiKV always consumes 2.21 GB DRAM to store B$^+$-Tree index, which is larger than FPTree. This

is because the fine-grained B$^+$-Tree of HiKV must index every unsorted key-value in NVM, while FPTree only stores its inner nodes in DRAM. However, the HiKV-ratio decreases from 40% to 4% as the value size increases from 64B to 1KB. For 256B value, the HiKV-ratio is 15.8%. Our evaluation shows that even though the extra DRAM space is used as a cache of FPTree (namely FPTree_C), HiKV still achieves higher performance than FPTree_C. Reducing the DRAM consumption of B$^+$-Tree, such as migrating part of leaf nodes to NVM, is our future work.

## 4.8 Recovery time

We finally evaluate the recovery performance of HiKV, NVStore, and FPTree. NVStore and FPTree takes 11.03s and 1.74s to recover 50M key-values, respectively. NVStore takes more time than FPTree as it allocates much larger contiguous inner nodes for tree index and insert keys more randomly than FPTree. Since the hash index is unsorted, HiKV needs to read valid index_entry in NVM and insert corresponding key and key-value position to the B$^+$-Tree index one by one. Thus, HiKV takes 88.24s to recover 50M key-values with one thread. However, increasing recovery threads allows to reduce the recovery time. For instance, HiKV takes 6.28s to recover 50M key-values with 16 threads.

# 5 Related Work

In this section, we discuss related works from three aspects: indexing structure, concurrency control, and NVM consistency guaranteeing.

**Indexing Structure.** Several distributed KV stores, such as Cassandra [43], Megastore [44], and SLIK [45], construct multiple indexes for multi-key-value data, such as secondary index for non primary key query. However, HiKV constructs a hybrid index according to a single key, and focuses on reducing the latency of updating hybrid index. SILT [18] and dual-stage index [19] construct multiple indexes to reduce DRAM consumption of indexes. These two techniques are orthogonal to HiKV to reduce the DRAM consumption of B$^+$-Tree.

NVM, especially PCM, suffers from limited write endurance. Thus, a number of research efforts are made to optimize the indexing structure for NVM to reduce writes to NVM [23, 25, 56, 57]. For example, Chen et al. [23] propose the unsorted leaf nodes of B$^+$-Tree to writes caused by sorting. Instead of focusing on reducing NVM writes, HiKV mainly optimizes indexing structure to support rich KV operations.

**Concurrency Control.** Concurrency control for multi-core processor has been widely studied in KV stores. Echo [54] and NVStore [28] use the MVCC for concurrency control. Chronos [58] and MICA [42] uses partitioning for concurrency control of hash tables. PALM [59] is lock-free concurrent B$^+$-Tree. FPTree adopts HTM to handle the concurrency of inner nodes, and fine-grained locks for the concurrency access of leaf nodes[30]. HiKV adopts similar techniques according to the features of hybrid index, which are partitioning for hash tables and HTM for B$^+$-Tree index.

**NVM consistency guaranteeing.** Recent research works propose techniques to reduce the cost of consistency guaranteeing. A few research works [60, 61, 62] use the differential logging [63] to only record modified bytes of a block on journal to reduce NVM writes. However, differential logging cannot avoid twice writes. Several works propose a combination of multiple techniques to reduce consistency cost according to data granularity. Atomic-write is used to update file system metadata, whose granularity is usually small such as 8B or 16B [5, 47, 64]. For large-granularity data, write-ahead logging and copy-on-write are used [5, 47]. NVStore [28], FPTree [30] also use ordered-write to guarantee consistency. However, HiKV adopts ordered-write accompanied with atomic-write to hash index, which can always achieve the minimum count of persists for different KV write operations.

# 6 Conclusion

Persistent key-value stores are widely deployed in real-world applications. Hybrid memory consisting of DRAM and NVM allows storage systems to persist data directly in the memory. Building KV stores towards hybrid memory can exploit its performance characteristic. Supporting rich KV operations (*Put/Get/Update/Delete/Scan*) efficiently is highly required by today's applications. However, either hash index or B$^+$-Tree index employed by existing KV stores cannot efficiently support all these operations. In this paper, we propose hybrid index to adopt a hash index in NVM for fast searching and directly persisting, and a B$^+$-Tree index in DRAM for fast updating and supporting range scan. On top of the hybrid index, we build HiKV, a hybrid index based key-value store with the well-performed scalability and crash consistency guaranteeing. Extensive experiments show that HiKV outperforms the state-of-the-art NVM-based KV stores.

# 7 Acknowledgments

We would like to thank the anonymous reviewers and our shepherd, Donald E. Porter, for their helpful comments. We also thank Ismail Oukid and Jun Yang for their help in figuring out the details of FPTree and NVStore, respectively. We thank Wenlong Ma for useful discussions. This work is supported by National Key Research and Development Program of China under grant No. 2016YFB1000302 , National Science Foundation of China under grant No. 61502448 and No. 61379042.

## References


[1] H. Wong, S. Raoux, S. Kim, J. Liang, J. Reifenberg, B. Rajendran, M. Asheghi, and K. Goodson, "Phase change memory," *Proceedings of the IEEE*, vol. 98, pp. 2201–2227, 2010.

[2] I. Baek, M. Lee, S. Seo, M. Lee, D. Seo, D.-S. Suh, J. Park, S. Park, H. Kim, and I. Yoo, "Highly scalable nonvolatile resistive memory using simple binary oxide driven by asymmetric unipolar voltage pulses," in *2004 IEEE International on Electron Devices Meeting*, IEDM'04, pp. 587–590, 2004.

[3] "Intel and Micron produce breakthrough memory technology." `https://newsroom.intel.com/news-releases/intel-and-micron-produce-breakthrough\-memory-technology/`.

[4] M. K. Qureshi, V. Srinivasan, and J. A. Rivers, "Scalable high performance main memory system using phase-change memory technology," in *Proceedings of the 36th annual international symposium on Computer architecture*, ISCA '09, pp. 24–33, 2009.

[5] X. Jian and S. Steven, "NOVA: A log-structured file system for hybrid volatile/non-volatile main memories," in *Proceedings of the 14th USENIX Conference on File and Storage Technologies*, FAST'16, pp. 323–338, 2016.

[6] J. Ou, J. Shu, and Y. Lu, "A high performance file system for non-volatile main memory," in *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys '16, pp. 12:1–12:16, 2016.

[7] 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," in *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '06, pp. 15–15, 2006.

[8] S. Ghemawat and J. Dean, "LevelDB." `https://leveldb.org`.

[9] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's highly available key-value store," in *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pp. 205–220, 2007.

[10] R. Sumbaly, J. Kreps, L. Gao, A. Feinberg, C. Soman, and S. Shah, "Serving large-scale batch computed data with project voldemort," in *Proceedings of the 10th USENIX conference on File and Storage Technologies*, FAST'12, pp. 18–18, 2012.

[11] Facebook, "RocksDB." `https://rocksdb.org`.

[12] D. Beaver, S. Kumar, H. C. Li, J. Sobel, and P. Vajgel, "Finding a needle in haystack: Facebook's photo storage.," in *Proceedings of the 9th Symposium on Operating Systems Design and Implementation*, OSDI'10, pp. 1–8, 2010.

[13] C. Lai, S. Jiang, L. Yang, S. Lin, G. Sun, Z. Hou, C. Cui, and J. Cong, "Atlas: Baidu's key-value storage system for cloud data," in *Proceedings of the 31st Symposium on Mass Storage Systems and Technologies*, MSST'15, pp. 1–14, IEEE, 2015.

[14] P. Shetty, R. P. Spillane, R. Malpani, B. Andrews, J. Seyster, and E. Zadok, "Building workload-independent storage with VT-trees," in *Proceedings of the 11th USENIX Conference on File and Storage Technologies*, FAST'13, pp. 17–30, 2013.

[15] L. Marmol, S. Sundararaman, N. Talagala, and R. Rangaswami, "NVMKV: A scalable, lightweight, ftl-aware key-value store," in *Proceedings of the 2015 USENIX Annual Technical Conference*, USENIX ATC'15, pp. 207–219, 2015.

[16] L. Lu, T. S. Pillai, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Wisckey: Separating keys from values in SSD-conscious storage," in *Proceedings of the 14th USENIX Conference on File and Storage Technologies*, FAST'16, pp. 133–148, Feb. 2016.

[17] B. Debnath, S. Sengupta, and J. Li, "Flashstore: High throughput persistent key-value store," *Proceedings of the VLDB Endowment*, vol. 3, pp. 1414–1425, Sept. 2010.

[18] H. Lim, B. Fan, D. G. Andersen, and M. Kaminsky, "SILT: A memory-efficient, high-performance key-value store," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pp. 1–13, 2011.

[19] H. Zhang, D. G. Andersen, A. Pavlo, M. Kaminsky, L. Ma, and R. Shen, "Reducing the storage overhead of main-memory oltp databases with hybrid indexes," in *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pp. 1567–1581, 2016.

[20] J. Levandoski, D. Lomet, and S. Sengupta, "The Bw-Tree: A b-tree for new hardware platforms," in *Proceedings of the IEEE 29th International Conference on Data Engineering*, ICDE'13, pp. 302–313, 2013.

[21] 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*, EuroSys '15, pp. 32:1–32:14, 2015.

[22] P. Wang, G. Sun, S. Jiang, J. Ouyang, S. Lin, C. Zhang, and J. Cong, "An efficient design and implementation of lsm-tree based key-value store on open-channel ssd," in *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, pp. 16:1–16:14, 2014.

[23] S. Chen, P. B. Gibbons, and S. Nath, "Rethinking database algorithms for phase change memory," in *Proceedings of the 5th Biennial Conference on Innovative Data Systems Research*, CIDR'11, pp. 21–31, 2011.

[24] S. Venkataraman, N. Tolia, P. Ranganathan, and R. H. Campbell, "Consistent and durable data structures for non-volatile byte-addressable memory," in *Proceedings of the 9th USENIX conference on File and stroage technologies*, FAST'11, pp. 5–5, 2011.

[25] J. Ni, W. Hu, G. Li, K. Tan, and D. Sun, "Bp-tree: A predictive b+-tree for reducing writes on phase change memory," *IEEE Transactions on Knowledge and Data Engineering*, vol. PP, no. 99, pp. 1–1, 2014.

[26] P. Chi, W.-C. Lee, and Y. Xie, "Making b+-tree efficient in pcm-based main memory," in *Proceedings of the 2014 International Symposium on Low Power Electronics and Design*, ISLPED '14, pp. 69–74, 2014.

[27] S. Chen and Q. Jin, "Persistent b+-trees in non-volatile main memory," *Proceedings of the VLDB Endowment*, vol. 8, pp. 786–797, Feb. 2015.

[28] 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 *Proceedings of the 13th USENIX Conference on File and Storage Technologies*, FAST'15, pp. 167–181, 2015.

[29] G. S. Choi, B. W. On, and I. Lee, "Pb+-tree: Pcm-aware b+-tree," *IEEE Transactions on Knowledge and Data Engineering*, vol. 27, no. 9, pp. 2466–2479, 2015.

[30] 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*, SIGMOD '16, pp. 371–386, 2016.

[31] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger, "Architecting phase change memory as a scalable dram alternative," in *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, pp. 2–13, 2009.

[32] K. Suzuki and S. Swanson, "The non-volatile memory technology database (NVMDB)," Tech. Rep. CS2015-1011, Department of Computer Science & Engineering, University of California, San Diego, 2015.

[33] M. F. Chang, J. J. Wu, T. F. Chien, Y. C. Liu, T. C. Yang, W. C. Shen, Y. C. King, C. J. Lin, K. F. Lin, Y. D. Chih, S. Natarajan, and J. Chang, "19.4 embedded 1mb reram in 28nm cmos with 0.27-to-1v read using swing-sample-and-couple sense amplifier and self-boost-write-termination scheme," in *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers*, ISSCC'14, pp. 332–333, Feb 2014.

[34] Micron, "SLC NAND flash products." `http://www.micron.com/products/nand-flash/slc-nand#fullPart,Dec.2014`.

[35] "MyRocks: A space- and write-optimized mysql database." `https://code.facebook.com/posts/190251048047090/myrocks-a-space-and-write-optimized-mysql-database/`.

[36] K. Ren and G. Gibson, "TABLEFS: Enhancing metadata efficiency in the local file system," in *Proceedings of the 2013 USENIX Annual Technical Conference*, USENIX ATC'13, pp. 145–156, 2013.

[37] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn, "Ceph: A scalable, high-performance distributed file system," in *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, pp. 307–320, 2006.

[38] K. Ren, Q. Zheng, S. Patil, and G. Gibson, "IndexFS: Scaling file system metadata performance with stateless caching and bulk insertion," in *Proceedings of the 2014 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC'14, pp. 237–248, 2014.

[39] Y. Mao, E. Kohler, and R. T. Morris, "Cache craftiness for fast multicore key-value storage," in *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, pp. 183–196, 2012.

[40] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden, "Speedy transactions in multicore in-memory databases," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pp. 18–32, 2013.

[41] "Memcached." `https://memcached.org`.

[42] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky, "MICA: A holistic approach to fast in-memory key-value storage," in *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'14, pp. 429–444, 2014.

[43] A. Lakshman and P. Malik, "Cassandra: a decentralized structured storage system," *ACM SIGOPS Operating Systems Review*, vol. 44, no. 2, pp. 35–40, 2010.

[44] J. Baker, C. Bond, J. C. Corbett, J. Furman, A. Khorlin, J. Larson, J.-M. Leon, Y. Li, A. Lloyd, and V. Yushprakh, "Megastore: Providing scalable, highly available storage for interactive services," in *Proceedings of the 5th Conference on Innovative Data system Research*, CIDR'11, pp. 223–234, 2011.

[45] A. Kejriwal, A. Gopalan, A. Gupta, Z. Jia, S. Yang, and J. Ousterhout, "SLIK: Scalable low-latency indexes for a key-value store," in *Proceedings of the 2016 USENIX Annual Technical Conference*, USENIX ATC'16, pp. 57–70, 2016.

[46] "Intel64 software developers manual (vol 2, ch 3.2)," 2013.

[47] S. R. Dulloor, S. Kumar, A. Keshavamurthy, P. Lantz, D. Reddy, R. Sankaran, and J. Jackson, "System software for persistent memory," in *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, pp. 15:1–15:15, ACM, 2014.

[48] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny, "Workload analysis of a large-scale key-value store," in *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE joint international conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '12, pp. 53–64, 2012.

[49] H. Volos, A. J. Tack, and M. M. Swift, "Mnemosyne: lightweight persistent memory," in *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems*, ASPLOS '11, pp. 91–104, 2011.

[50] X. Wu and A. L. N. Reddy, "Scmfs: a file system for storage class memory," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pp. 39:1–39:11, 2011.

[51] "Intel architecture instruction set extensions programming reference." `https://software.intel.`

com/sites/default/files/managed/69/78/319433-025.pdf.

[52] J. Huang, K. Schwan, and M. K. Qureshi, "Nvram-aware logging in transaction systems," *Proceedings of the VLDB Endowment*, vol. 8, no. 4, pp. 389–400, 2014.

[53] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with ycsb," in *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pp. 143–154, 2010.

[54] 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*, INFLOW '13, pp. 4:1–4:8, 2013.

[55] Y. Zhang and S. Swanson, "A study of application performance with non-volatile main memory," in *Proceedings of the 31st Symposium on Mass Storage Systems and Technologies*, MSST'15, pp. 1–10, IEEE, 2015.

[56] B. Debnath, A. Haghdoost, A. Kadav, M. G. Khatib, and C. Ungureanu, "Revisiting hash table design for phase change memory," *ACM SIGOPS Operating System Review*, vol. 49, pp. 18–26, Jan 2016.

[57] P. Zuo and Y. Hua, "A write-friendly hashing scheme for non-volatile memory systems," in *Proceedings of the 33st Symposium on Mass Storage Systems and Technologies*, MSST'17, pp. 1–10, 2017.

[58] R. Kapoor, G. Porter, M. Tewari, G. M. Voelker, and A. Vahdat, "Chronos: Predictable low latency for data center applications," in *Proceedings of the Third ACM Symposium on Cloud Computing*, SoCC '12, pp. 9:1–9:14, 2012.

[59] J. Sewall, J. Chhugani, C. Kim, N. Satish, and P. Dubey, "PALM: Parallel architecture-friendly latch-free modifications to b+ trees on many-core processors," *Procedings of the VLDB Endowment*, vol. 4, no. 11, pp. 795–806, 2011.

[60] E. Lee, S. Yoo, J.-E. Jang, and H. Bahn, "Shortcut-jfs: A write efficient journaling file system for phase change memory," in *Proceedings of the IEEE 28th Symposium on Mass Storage Systems and Technologies*, MSST'12, pp. 1–6, 2012.

[61] J. Chen, Q. Wei, C. Chen, and L. Wu, "FSMAC: A file system metadata accelerator with non-volatile memory," in *Proceedings of the IEEE 29th Symposium on Mass Storage Systems and Technologies*, MSST'13, pp. 1–11, IEEE, 2013.

[62] W.-H. Kim, J. Kim, W. Baek, B. Nam, and Y. Won, "Nvwal: Exploiting nvram in write-ahead logging," in *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, pp. 385–398, 2016.

[63] J. Lee, K. Kim, and S. K. Cha, "Differential logging: A commutative and associative logging scheme for highly parallel main memory database," in *Proceedings of the 17th International Conference on Data Engineering*, ICDE'01, pp. 173–182, IEEE, 2001.

[64] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. Lee, D. Burger, and D. Coetzee, "Better I/O through byte-addressable, persistent memory," in *Proceedings of the Twenty-Second ACM Symposium on Operating Systems Principles*, SOSP '09, pp. 133–146, 2009.