FlatStore: An Efficient Log-Structured Key-Value Storage Engine for Persistent Memory

Youmin Chen[♭†], Youyou Lu[♭†], Fan Yang[♭†], Qing Wang[♭†], Yang Wang[‡], Jiwu Shu[♭†*]
Department of Computer Science and Technology, Tsinghua University[♭]
Beijing National Research Center for Information Science and Technology (BNRist)[†]
The Ohio State University[‡]

Abstract

Emerging hardware like persistent memory (PM) and high-speed NICs are promising to build efficient key-value stores. However, we observe that the small-sized access pattern in key-value stores doesn't match with the persistence granularity in PMs, leaving the PM bandwidth underutilized. This paper proposes an efficient PM-based key-value storage engine named FlatStore. Specifically, it decouples the role of a KV store into a persistent log structure for efficient storage and a volatile index for fast indexing. Upon it, FlatStore further incorporates two techniques: 1) compacted log format to maximize the batching opportunity in the log; 2) pipelined horizontal batching to steal log entries from other cores when creating a batch, thus delivering low-latency and high-throughput performance. We implement FlatStore with the volatile index of both a hash table and Masstree. We deploy FlatStore on Optane DC Persistent Memory, and our experiments show that FlatStore achieves up to 35 Mops/s with a single server node, 2.5 - 6.3 times faster than existing systems.

CCS Concepts • **Information systems → Information storage systems**.

Keywords key-value store, log structure, persistent memory, batching

ACM Reference Format:
Youmin Chen, Youyou Lu, Fan Yang, Qing Wang, Yang Wang, Jiwu Shu. 2020. FlatStore: An Efficient Log-Structured Key-Value Storage Engine for Persistent Memory. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'20), March 16–20, 2020, Lausanne, Switzerland.* ACM, NY, NY, USA, 15 pages. https://doi.org/10.1145/3373376.3378515

*Jiwu Shu is the corresponding author.

ASPLOS '20, March 16–20, 2020, Lausanne, Switzerland

ACM ISBN 978-1-4503-7102-5/20/03…$15.00
https://doi.org/10.1145/3373376.3378515

1 Introduction

Key-value store with the simple abstraction of interfaces (i.e., *Put*, *Get*) has become a fundamental component in datacenter infrastructures [14, 21, 38, 45]. Recently, the increasing demands in fast data storage and processing invoke the unique question on *how to design efficient key-value stores to handle write-intensive and small-sized workloads*, which is required in today's production deployments [8, 21, 34, 37, 45, 48, 54].

Fortunately, emerging networking and storage technologies, such as InfiniBand and persistent memory (PMs), bring the new opportunity to achieve this goal: PCM [35, 50, 63], ReRAM [9], and the recently released Optane DC Persistent Memory [6] (abbreviated as "Optane DCPMM" in the rest of the paper), are capable of providing data persistence while achieving comparable performance and higher density than DRAM. Similarly, recently released 100 Gbps InfiniBand network also delivers sub-microsecond latency and extremely high message rates [5]. This work studies how to maximize the performance of key-value stores with the help of such new devices. It is motivated by our in-depth analysis of many recent KV stores [15, 26, 27, 44, 47, 59, 61, 64], whose access pattern doesn't match with the access granularity in PMs.

Both our analysis and previous works [47, 49, 61] report that KV stores generate a huge number of small-sized writes (e.g., a large proportion of KV pairs contain only several or tens of bytes in the production workloads [8, 37, 45]). Furthermore, the index structure inside a KV store causes a great amplification of writes. For a hash-based index, multiple entries need to be rehashed if their keys conflict or the hash table is resized [64]. A tree-based index also needs to frequently shift the index entries in each tree node to keep them ordered, and merge/split the tree nodes to keep the tree balanced [15, 27, 47, 61]. Most of such updates only involve updating one or several pointers. For failure atomicity, the updated data needs to be explicitly persisted with extra flush instructions (e.g., `clflushopt/clwb`). However, CPUs flush data at cacheline granularity (64 B in AMD and x86 platform), and PM may have even coarser internal block size (e.g., 256 B in Optane DCPMM [29]), which is larger than the write sizes in most cases, wasting the hardware bandwidth dramatically. We deploy a persistent KV index named FAST&FAIR [27] on

Optane DCPMM, and observe that it achieves 3.5 Mops/s for *Put* operations, which is merely 6% the raw throughput of Optane DCPMM.

A classic approach to address the above problems is to manage the KV store with a log structure [36, 46, 52], so all the updates can be simply appended to the log. More critically, by batching multiple requests from the clients and performing the updates together, the write overhead is amortized across multiple requests. Note that the performance benefit of batching is limited by the number of updates we can perform together. Thus, this idea is very successful for HDD or SSD since they have better sequential write performance, and each batch can contain multiple consecutive sectors (up to tens of MBs). However, its application in NVM-based systems is more challenging: 1) Our measurements reveal that, as long as the I/O size is larger than the minimal I/O unit (i.e. 256B block size) and there are enough threads to perform I/Os concurrently, sequential and random writes to Optane DCPMM have very similar bandwidth, so it is not beneficial to batch more data than a single I/O unit. Moreover, PM has much finer access granularity (i.e., 64 B flush size and 256 B block size), which can only accommodate a very limited number of log entries if we simply record every memory updates in the log. 2) Batching increases latency inevitably, preventing its adoption in emerging PMs and high-speed NICs, since low latency is one of their major benefits. Because of these challenges, we find, although some recent works [25, 60] apply the log-structured idea in PM-based storage systems, they mainly adopt the log structure to ensure failure atomicity or reduce memory fragmentation, but do not explore the opportunities of batching to amortize the persistence overhead.

This paper proposes FlatStore, a persistent memory KV storage engine, to revitalize the log-structured design in PMs. The key idea is to decouple the role of a KV store into a *volatile index* for fast indexing and a *persistent log structure* for efficient storage. *Compact log format* and *pipelined horizontal batching* techniques are then introduced to address the challenges mentioned above, so as to achieve high throughput, low latency, and multi-core scalability.

Compact log format is introduced to address the first challenge (i.e., improve batching chance). FlatStore only stores index metadata and small KVs in the persistent log. Large KVs are stored separately with an NVM allocator, because they cannot benefit from batching in the log anyway. We then use the operation log technique [12] to format the log entries, which simply describes each operation, instead of recording every index updates, so as to minimize the space consumption of the index metadata. Note that we don't need to persist the allocation metadata (i.e., bitmap) within the NVM allocator, since we observe that there is a redundancy between the index metadata in the log and the metadata within the NVM allocator: the KV index maps a key to the address of the KV on NVM; the NVM allocator records all the allocated addresses, which are exactly the addresses of the KVs on NVM. By using *lazy-persist allocator* to manage the NVM space for each allocation, we can deterministically reconstruct the allocation metadata in the NVM allocator by scanning the log after rebooted. In this way, FlatStore restricts the size of each log entry to a minimum of 16 bytes, enabling multiple log entries to be flushed together to PMs.

To address the second challenge (i.e., reduce latency when batching), FlatStore adopts a "semi-partitioned" approach. It still leverages the existing per-core processing and user-level polling mechanism [23, 30] when serving the client requests, to avoid high software overhead and lock contention, but incorporates a novel technique named *Pipelined Horizontal Batching* to persist the log entries. It allows a server core to collect log entries from other cores when creating a batch: compared to letting each core create batches from its own requests, this approach significantly reduces the time to accumulate enough log entries before flushing them to PM. To mitigate the contention among the cores, pipelined horizontal batching 1) carefully schedules the cores to release the lock in advance while without compromising the correctness, and 2) organizes the cores into groups to balance the contention overhead and batching opportunity.

We implement FlatStore with in-memory index structure of both a partitioned hash table (FlatStore-H) and Masstree [41] (FlatStore-M). Our evaluation on Optane DCPMM shows that they can respectively handle up to 35 and 18 million *Put* operations per second with one server node, 2.5 - 6.3× faster than existing systems.

2 Background and Motivation

In this section, we perform an extensive analysis on the production key-value store workloads, and describe the mismatch between the small-sized access pattern in existing KV stores and the persistence granularity of PMs.

2.1 Production Key-Value Store Workloads

A large portion of items stored and manipulated by KV stores in production environments are small-sized. For instance, the Facebook ETC Memcached pool has 40% of items with sizes smaller than 13 bytes, and 70% of items with sizes smaller than 300 bytes [8]. Rajesh et al. [45] reported that 50% of items in three Facebook pools are smaller than 250 bytes. Many popular in-memory computation tasks, such as sparse parameters in linear regression and graph computing, typically store small-sized key-value items [37].

Shifting from read-dominated to write-intensive. Historically, key-value stores such as Memcached are widely used as an object caching system, and there are much more reads than writes [8]. However, a key-value store today must also handle write-intensive workloads, e.g., the frequently-changing objects [7], the popular e-commerce platform generating new

transactions at an extremely fast rate [21], and the emerging serverless analytics exchanging short-lived data [34].

2.2 Small Updates Mismatch with Persistence Granularity

A typical persistent memory key-value store mainly consists of two parts: 1) a *persistent index* (e.g., tree, hash table) to find the key-value records and 2) a *storage manager* (i.e., NVM allocator [3, 11]) to store the actual KV pairs. These systems store KV records in NVM but store index in various ways: some of them (e.g., CDDS-Tree [57], wB$^+$-Tree [15], FAST&FAIR [27], CCEH [44] and Level-Hashing [64]) store the whole index in NVM; some of them (e.g. FPTree [47] and NV-Tree [61]) only store the leaf nodes of the index tree in NVM but store the inner nodes directly in DRAM; some of them (e.g. Bullet [26] and HiKV [59]) manage two mirrored indexes (one in DRAM and the other in NVM) and use backend threads to synchronize them. In such designs, a put operation usually involves multiple updates to NVM, including ① an update to the actual KV, ② an update to the metadata inside the allocator, and ③ multiple updates to the index. For ①, a large proportion of KV pairs in production workloads are small-sized. For ③, *updating the index structure often causes a huge amplification of NVM writes*: A hash table needs to move the index entry to its alternative slot when the two keys conflict, and rehash the buckets when the hash table is resized. Tree-based indexes also need to shift the entries in each leaf node to keep them sorted, and merge/split the tree nodes to keep the tree balanced. Most of such updates only involve modifying one or several pointers.

Optimizations of modern processors and compilers may reorder the store operations from CPU cache to the NVM. Hence, to prevent data from being lost or partially updated in case of system failure, the KV store needs to either carefully order updates or log updates first, which need to be flushed from volatile CPU caches to NVM. Cacheline flush instructions (e.g., *clflush*, *clflushopt* and *clwb*) and memory fence instructions (e.g., *mfence* and *sfence*) are used to enforce such ordering. However, these flush instructions flush data with cacheline granularity, which is typically 64 B, and PM has even coarser internal write granularity (256 B in Optane DCPMM [29]). Hence, there is a mismatch between the write size in KV stores and the update granularity in the hardware, wasting the PM bandwidth dramatically.

We investigate such persistence problem by analyzing FAST&FAIR [27] on Optane DCPMM. FAST&FAIR is an efficient and persistent B$^+$-Tree which completely avoids the logging overhead by transforming a B$^+$-Tree to another consistent state or a transient inconsistent state that readers can tolerate. We use *Put* operations with 8-byte values and measure the throughput by increasing the number of threads. For comparison, we also measure the raw performance of Optane DCPMM by issuing 64-byte random writes. Our platform is equipped with 4 Optane DCPMM Persistent Memory

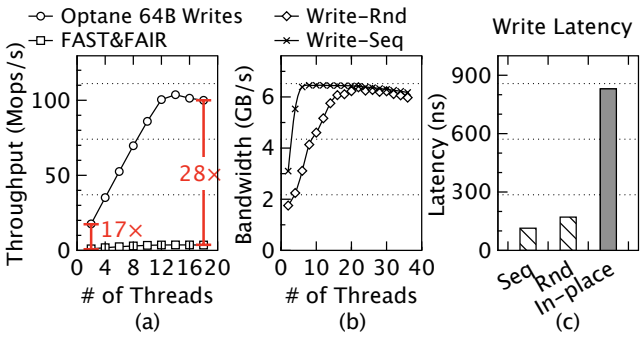

Figure 1. Performance Evaluation of Intel Optane DCPMM.

DIMMs (256 GB per DIMM, 1 TB in total) and two Intel Xeon Gold 6240M CPUs (36 cores in total). More detailed hardware configuration is shown in Section 5. From Figure 1(a), we observe that the throughput of Optane DCPMM is 17× higher than FAST&FAIR. The most important reason is that FAST&FAIR needs to generate multiple small-sized writes for each operation, wasting PM bandwidth unnecessarily. Such performance gap becomes bigger as the number of threads increases, since Optane DCPMM has non-scalable write bandwidth.

2.3 Empirical Study on Optane DCPMM

Izraelevitz et al. [29] has already conducted an in-depth performance review of Optane DCPMM and observed some interesting findings. In our paper, we extend their measurements and make some new observations, which is helpful for the design of FlatStore. In our evaluation, each write operation is followed by `clwb` and `mfence` instructions, ensuring that the written data has reached Optane DCPMM (we omit the `pcommit` instruction, as Intel recommends).

(1) *Sequential and random access patterns have similar bandwidth under high concurrency.* Figure 1(b) summarizes the sequential and random performance of 256 B writes. With less than 20 concurrent threads, we can observe that the bandwidth of Optane DCPMM is 2× or higher when accessed in a sequential pattern. However, their bandwidth becomes the same when there are more threads. From a hardware perspective, sequential but concurrent access to the device actually becomes random accesses, since each thread writes to different addresses. Note that the figures we measured here are lower than the actual bandwidth (i.e., 8.6 GB/s) because we add extra flush instruction for each write operation.

(2) *Repeat flush to the same cacheline is delayed dramatically.* We observe that when a write operation is flushed (via `clwb`), a following write and flush to the same cacheline will be blocked for almost 800 ns (see In-place operation in Figure 1(c)). We suspect that there are two possible reasons: 1) `clwb` is issued asynchronously, and the latter `clwb` is blocked until the first one is finished and such acknowledgement is

not returned in time. 2) Optane DCPMM may have the on-chip wear-levelling function, which blocks the latter flush when they access the same cacheline. Anyway, such behavior is a big problem for those storage systems with "in-place" update manner, especially when running a skewed workload.

2.4 Challenges

A classic solution to address such mismatch is to incorporate the log structure, which forms sequential writes by always appending updates to the tail of the log. Log structure also allows us to batch multiple small updates. Thus we only need to pay the overhead of flushing data once. However, its application in NVM-based systems is more challenging:

Log-structured storage has less batching opportunities in PM. Log structure is widely used in HDD and SSD-based storage systems, since appending data to the end of the log forms sequential writes, which are friendly to HDD and SSD. What's more, both of them have much coarser flushing granularity (e.g., 4 KB), so a storage system can buffer up to tens of MBs of data before flushing them. However, as analyzed before, Optane DCPMM shows similar performance between sequential and random accesses under high concurrency, so a KV store cannot benefits from sequential access anymore. PMs also have much finer access granularity (64 B flush size and 256 B internal block size), which limits the number of updates we can flush together. Even worse, log structure increases the size of each update, since each log entry needs to encapsulate extra metadata. For example, many systems incorporate a bitmap to keep track of the unused space. Updating the bitmap in place only changes one bit; logging such updates, however, needs to record the address, which is usually 8B. Such amplification makes it harder to benefit from batching.

Batching increases latency. Compared to traditional Gigabyte Ethernet and HDD/SSD, low latency is one of the major benefits of emerging NVMs and high-speed NICs. However, batching requests naturally increases the latency because the system needs to accumulate multiple requests before processing them. Meanwhile, batching also conflicts with the design principles of high-speed networks, which suggests user-level polling and independent per-core processing to avoid contention caused by shared request/thread pools—this reduces the chances of batching requests and may further increase the latency with imbalanced workloads.

3 FlatStore Design

3.1 Overview of FlatStore

This paper proposes FlatStore, a persistent key-value storage engine with three key design principles: *minimal write overhead*, *low latency*, and *multi-core scalability*. Figure 2 summarizes its high-level design components:

- *Compacted OpLog* (Section 3.2). FlatStore incorporates a per-core log-structured OpLog to absorb frequent and small-sized updates with efficient batching. To maximize the chance of batching, FlatStore compacts each log entry by only storing metadata and small KV pairs in it. Large KVs are stored separately with an allocator, because they cannot benefit from logging.
- *Lazy-persist Allocator* (Section 3.2). It is proposed to store large KV items. Since the OpLog already records the addresses of large KVs, it's unnecessary to flush the allocation metadata (i.e., bitmap) anymore. We remove such redundancy by using a multi-class based allocation policy and lazily persisting the allocation metadata during the runtime. After a crash, the bitmap can be recovered by calculating the offset with the pointers in the OpLog.
- *Pipelined Horizontal Batching* (Section 3.3). In Flatstore, clients send their requests to server cores through a customized RDMA-based RPC (Section 4.3) and the server cores are determined by the keyhashes. Hence, server cores can persist log entries to their local OpLogs in parallel. To maximize the chance of batching, FlatStore incorporates Pipelined Horizontal Batching to allow a core to steal log entries from other cores when creating a batch.

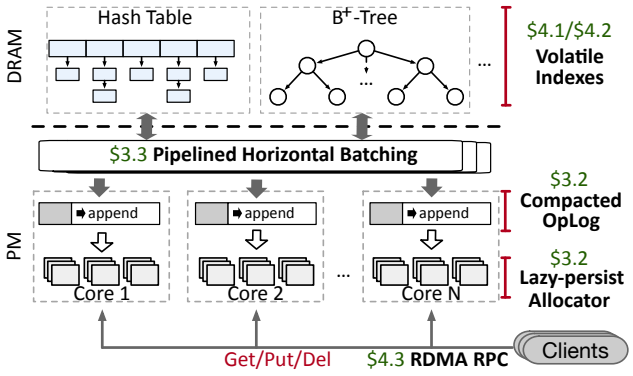

Figure 2. Architecture of FlatStore.

FlatStore maintains an extra copy of volatile index in DRAM, so as to avoid the server cores from scanning the whole OpLog when serving the *Get* requests. FlatStore can use any existing index solutions. In our implementation, we respectively deploy FlatStore with ❶ a partitioned hash table for fast indexing (Section 4.1) and ❷ a global Masstree [41] for efficient range searching (Section 4.2).

3.2 Compacted OpLog and Lazy-Persist Allocator

As described before, updating in a persistent key-value store often causes a huge amplification of writes and cache flushing. To address such issue, FlatStore decouples the key-value store into a *volatile index* (by directly using existing index schemes) for fast indexing, a *compacted per-core OpLog* to absorb small updates, and a *persistent allocator* for storing large KVs. To process an update request (i.e., *Put*/*Del*), the server core simply writes the key-value item, appends a log entry

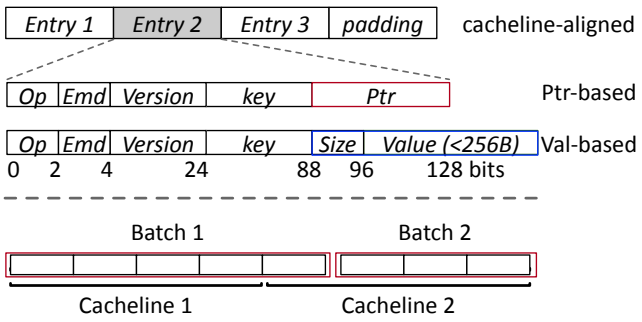

Figure 3. Layout of the Log Entry with Two Cases (pointer-based entry only stores a pointer, while value-based entry has varied sizes with 1 - 256 B values).

at the end of its local log, and then updates the volatile index accordingly. Thus, the persistence overhead caused by re-hashing as in hash index and shifting/splitting/merging as in tree index is avoided. To serve a *Get* request, the server core locates a key-value item by firstly referring to the volatile index with the given key to find the exact log entry, and finally to the key-value item. In this way, the overhead of scanning the whole log to find a specific key is avoided too.

Log Entry Compaction. A key design aspect of OpLog is how to construct the layout of the log entry: the log entry should be designed small enough to better support batching, enabling FlatStore to persist more log entries within one flush operation. Therefore, FlatStore only places the index metadata and extremely small-sized KV items in OpLog (256 B in our implementation, which is enough to saturate the bandwidth of Optane DCPMM), while other large key-value items are stored separately with an allocator. Meanwhile, each log entry also needs to contain sufficient metadata for normal indexing at runtime and to safely recover the lost volatile index structure after system crashes.

We propose to incorporate the operation log technique [12] to further compact the log entries: Instead of recording each memory updates, the log entry only contains minimal information for describing each operation. As shown in Figure **??**, each log entry consists of five parts, among them, *Op* records the operation type (i.e., *Put* or *Delete*), *Emd* means whether the key-value item is placed at the end of the log entry, and *Version* is introduced to guarantee the correctness of log cleaning (in Section 3.4). Similar to existing works [26, 47, 61], we use 8-byte keys. Note that FlatStore can place the keys out of the OpLog to support larger keys, as we do with the values. *Ptr* is used to point to the actual records when it is stored out of the OpLog. Otherwise, the value, as well as its size, are placed directly at the end of the log. To minimize the log entry size, we only reserve 40 bits for the *Ptr*. This is acceptable since the allocator only manages large KVs, whose sizes are larger than 256 bytes, so the lower 8 bits are dismissed (40+8 bits of pointers are capable of indexing 128 TB of NVM space). Therefore, the size of each log entry is restricted to be 16 bytes (for pointer-based entries), indicating that 16 log entries can be flushed altogether, with the overhead equal to that of persisting a single log entry.

Padding. In FlatStore, in-place flushes to the same cacheline (described in Section 2.3) are avoided in most cases, since all the data is "out-of-place" updated. However, two adjacent batches may still share the same cacheline in the OpLog, since not every time we collect exactly 16 log entries when building a batch (as shown at the bottom of Figure **??**). Under the circumstances, the persisting of the latter batch is delayed. To solve this problem, we add padding at the end of each batch to make them cacheline-aligned. This is quite difficult for a storage system with "in-place" update manner to deal with, especially when running a skewed workload.

Lazy-persist Allocator. Storing large KVs with an allocator causes extra flushing overhead, since the introduced NVM allocator also needs to carefully maintain its private metadata, tracking which addresses in the NVM are used and which are free. However, we observe that there is a redundancy of flushing between the log entry and the allocation metadata: Once a record has been successfully inserted, the *Ptr* in the log entry always points to an allocated data block storing this record. Hence, we can lazily persist the allocation metadata and correctly recover them after a system failure leveraging such redundant information. The main challenge is how to use the *Ptr* in each log entry to reversely locate the allocation metadata during the recovery, since the allocator is required to support variable-length allocation.

We propose a Hoard-like [10] *lazy-persist allocator*. It first cuts the NVM space into 4 MB chunks. Further, the 4 MB chunks are cut into different classes of data blocks, and the data blocks in the same chunk have the same class of size. Such cutting size is persistently recorded at the head of each NVM chunk when it is ready for allocation. A bitmap is also placed at the head of each chunk to track the unused data blocks. With such design, the starting address of each chunk is 4MB-aligned and the allocation granularity of each chunk is specified at the head of it. Therefore, the offset of an allocated NVM block in the chunk can be calculated directly with *Ptr* in a valid log entry, enabling us to recover the bitmap even when they fail to be persisted before system crashes. Considering the scalability issue, these 4 MB NVM chunks are partitioned to different server cores. Upon receiving an allocating request, the allocator first chooses a proper class of NVM chunk from its privately managed NVM chunks, and then allocates a free data block by modifying the bitmap (without flushing). For an allocation with size larger than 4 MB (which is less likely to happen in a key-value store), we directly assign it with one or multiple contiguous chunks.

Putting everything together, a normal *Put* request is processed at the server-side with the following steps:

1) Allocate a data block from the *lazy-persist* allocator, copy the record into it with the format of $(v\_len, value)$ and persist it. (This step is skipped for small KV record)

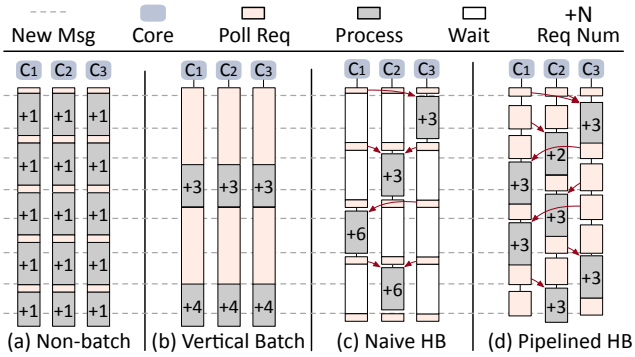

Figure 4. Sequence Diagram of Different Execution Model (gray dashed line indicates a new request arrives, "+N Req Num" means N of requests are batched together for persistence, red arrow means the action of stealing).

2) Initialize a log entry with each field filled: *Ptr* points to the block in step 1 if the record is placed out of the OpLog, and the *Version* is increased by one if this record already exists. Then, append the log entry to the log and persist it. Finally, update the tail pointer to point to the tail of the log and persist it.
3) Update the corresponding entry in the volatile index to point to this log entry.

For crash consistency, we need to "out-of-place" update a record if it already exists (instead of overwriting). Therefore, except for the above three steps, we need to free the old data block once the inserting finishs. Note that the freed data block can be reused immediately since the "read-after-delete" anomaly doesn't occur in FlatStore: the KVs are forwarded to a specific core according to the keyhash, and the operations with the same key go to the same core and are serialized by a conflict queue (see Section 3.3). A *Put* operation is considered persistent only after step 2 is finished. If a system failure occurs after step 2, FlatStore can still recover the volatile index and the allocation bitmap by replaying the OpLog (Section 3.5). To process the *Delete* operation, we simply append a tombstone log entry like RAMCloud [53].

We can observe that each *Put* operation in FlatStore only concerns **three** flush operations (the KV record, log entry and the tail pointer). By directly storing the small-sized record in the log entry, the number of flushing is further reduced. Selectively placing the KV records in OpLog and allocator also supports more efficient garbage collection [60]. FlatStore needs to periodically reclaim the NVM spaces when the log entries in OpLog become obsolete (Section 3.4). By only placing the 16-byte log entries and tiny KV items in the OpLog, the log compaction doesn't occupy CPU resources too much.

3.3 Horizontal Batching

The OpLog is designed to better support batching. Following this way, the server core can receive multiple client requests

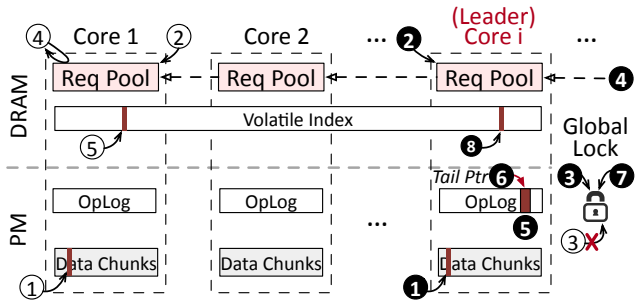

Figure 5. FlatStore with Horizontal Batching (The leader executes the steps with black circled numbers, while others follow the white circled ones).

from the network and process them all together. Assuming that N *Put* requests arrive, FlatStore first allocates the data blocks to store and persist the KV items for each of the requests. It then merges their log entries and flushes them together to the OpLog. Finally, the in-memory index is updated accordingly. With batching, the number of PM writes is reduced from original 3N to N + 2. Two reasons account to such reduction: (1) With the design of append-only OpLog, the log entries from different requests get the chance to be flushed together; (2) Updating of the *Tail Pointer* of the OpLog occurs from once-per-request to once-per-batch. For small KV items that are directly stored in the operation log, the persist operations can be further reduced.

Since the above batching scheme only batches on the requests received by each server core, we call this approach as *vertical batching*. *Vertical batching* reduces the persisting overhead, but it introduces the side-effects of higher latency: it reduces the chance of batching per core, so each core takes longer to accumulate enough requests. Figure 4 (a) and (b) compare the effects of (non-)batching and we can observe that *vertical batching* increases the response latency significantly. However, the sub-microsecond hardware (e.g., RDMA and NVM) drives us to achieve low-latency request processing [48, 54]. To this end, we propose *horizontal batching*. It allows a server core to steal the to-be-persisted log entries from other cores when building a batch.

Pipelined Horizontal Batching (Pipelined HB). As shown in Figure 5, to steal the log entries from other cores, we introduce 1) a global lock to synchronize the cores, and 2) a per-core request pool for inter-core communication. A *Put* operation is decoupled into three phases, which are *l-persist*, *g-persist* and *volatile* phase respectively. Space allocation and persisting of KV items are finished in *l-persist* phase, persisting of log entries is done in *g-persist* phase, and the update of volatile index lies in the *volatile* phase.

The steps of a naive HB approach is shown in Figure 5. After finishing the *l-persist* phase (❶/①), each CPU core puts the addresses of the to-be-persist log entries in local request pool (❷/②), and tries to acquire the global lock (❸/③).

The core that successfully acquires the lock becomes the *leader* (i.e., Core i), while the others act as followers and wait for the completion of the posted requests (④). The *leader* then fetches the log entries from other cores (❹). We simply let the leader scan all the cores and fetch all exiting log entries, instead of using the more complicated strategy (e.g., timeout or other threshold settings), to reduce latency. The collected log entries are merged together and appended to the OpLog in a batch manner (❺ and ❻). After this, it releases the global lock (❼) and informs other cores of the completion of persisting. Finally, both the *leader* and the followers update the in-memory index and send the response messages to the clients (❽/⑤).

We can observe that the naive approach inherits the property of batching by stealing the requests. However, it is sub-optimal since the processing of the three phases is strictly ordered and most CPU cycles are spent on waiting for the completion of log persisting (Figure 4 (c) explains this microscopically). Accordingly, we further propose *Pipelined HB* to interleave the execution of each phase: Once a server core fails to acquire the global lock, it turns to poll the arrival of the next requests and execute the logic of a second horizontal batching. These followers only asynchronously wait for the completion message from the previous *leader*s. Besides, the *leader* releases the global lock as soon as it has collected the log entries from other cores. Therefore, the log persisting overhead is moved out of the global lock, and adjacent HBs can be processed in parallel (shown in Figure 4 (d)).

Discussion. Pipelined HB may reorder clients' requests: When a server core processes a sequence of requests, the latter *Get* operation may fail to see the effect of a previous *Put* with the same key. Such case happens when the *Put* is still processed by the *leader*, and the server core switches to process the following requests. To address this issue, each server core maintains an exclusively owned conflict queue to track the requests being served. Any later requests will be delayed if their keys conflict.

Pipelined HB is similar to work stealing in stealing log entries from other cores, however, they work in a different way: Work-stealing rebalances the loads by letting the idle thread steal requests from busy ones. Pipelined HB relies on work-stealing, but works in the opposite way by letting one core get all request, to reduce latency in batching.

Pipelined HB with Grouping. On a multi-socket platform, acquiring the global lock by a large number of CPU cores leads to significant synchronization overhead. To address this issue, the server cores are partitioned into groups to execute pipelined HB. Therefore, a proper group size balances the global synchronization overhead and the batch size: Smaller group size incurs low locking overhead, with the cost of decreased size of each batch, or conversely. Based on our empirical results, arranging all the cores from the same socket into one group provides the optimal performance.

3.4 Log Cleaning

To avoid the length of the OpLog from growing arbitrarily, we need a method to compact it and drop those obsolete log entries. In FlatStore, each server core maintains an in-memory table to track the usage of each 4MB chunk in the OpLog when processing the normal *Put* and *Delete* requests. Whether a block is inserted into a reclaim list depends on both the ratio of live entries in it and the total number of free chunks. Each HB group launches one background thread to clean the log (denoted as the cleaner). Thus, log recycling is executed in parallel between different groups. The cleaner periodically checks the reclaim list to find the victim chunks. To recycle a NVM chunk, the cleaner first scans the chunk to determine the liveness of each log entry. This is achieved by comparing the *Version* of the log entry with the newest one in the in-memory index. Then, all the live log entries are copied to a newly allocated NVM chunk. Identifying the liveness of tombstones (for *Del*) is more complicated [53], which can be safely reclaimed only after all the log entries related to this KV item have been reclaimed. After that, the cleaner updates the corresponding entries in the in-memory index to point to their new locations with atomic CAS. Finally, the old block is freed by putting it back to the allocator. When the new NVM chunk is filled, it then links this block to the OpLog accordingly. The address of the new NVM chunk also needs to be tracked, to prevent it from being lost in case of system failures. We record such addresses in a journal field (a predefined area in PM), which can be reread during the recovery.

3.5 Recovery

We describe the recovery of FlatStore after a normal shutdown and system failure in this section.

Recovery after a normal shutdown. Before a normal shutdown, FlatStore copies the volatile index to a predefined location of NVM, and flushes the bitmap of each NVM chunk as well. Finally, FlatStore writes a *shutdown* flag to indicate a normal shutdown. After a reboot, FlatStore firstly checks and reset the state of this flag. If the flag indicates a normal shutdown, FlatStore then loads the volatile index to DRAM.

Recovery after a system failure. If the *shutdown* flag is invalid, the server cores need to rebuild the in-memory index and the bitmaps by scanning their OpLogs from the beginning to end. The NVM chunks tracked by the journal (mentioned in Section 3.4) are scanned too. The *Key* of each scanned log entry is firstly used to locate the slot in the volatile index. A new entry is inserted in the volatile index if we cannot find such a slot. Otherwise, it judges whether or not to update the pointer of this index entry by further comparing the *Version* number. With *Ptr* field, the allocation metadata (i.e., bitmaps) is set as well. Based on such design, FlatStore only needs to sequentially scan the OpLog to recover all the volatile data structures. In our experiments, it

only takes 40 seconds to recover 1 billion KV items. Such recovery time is tolerable since: 1) Many production workloads [8, 45] typically have a wide distribution of value sizes (small values dominate in terms of number while large ones dominate in space consumption) and thus have less index to recover in normal cases. 2) To shorten such recovery time, FlatStore also supports to checkpoint the volatile index into PMs periodically when the CPU is not busy.

4 Implementation

4.1 FlatStore-H: FlatStore with Hash Table

FlatStore can use any existing indexes as its volatile index. In our implementation, we deploy CCEH [44] in DRAM to work as the volatile index. Specifically, CCEH is partitioned into ranges with separate keyhash, and each core owns a CCEH instance. Hence, they can modify the hash table without any locking overhead. Since the index persistence has already been guaranteed by the OpLog, we place CCEH directly in DRAM and remove all its flush operations. A client will directly send a request to the specific core that is responsible for the key of this request (through a customized RPC in Section 4.3). Each core in FlatStore only updates its own volatile index, but persists log entries with pipelined HB. Each bucket in CCEH contains multiple index slots, including ① an array of *Keys* and co-located *Versions* to distinguish different KV items ② and an array of pointers pointing to the log entries in the OpLog.

The partitioned design can lead to load imbalances for skewed access patterns. However, horizontal batching is able to largely mitigate such imbalance: It spreads the load of persisting log operations, which are the most time-consuming part, between the cores. Our experiments also exhibit that FlatStore is good at handling skewed workloads.

4.2 FlatStore-M: FlatStore with Masstree

Since many existing persistent key-value stores [15, 47, 57, 61] build tree structures to support ordered/range searching, we also implement FlatStore with a tree-based index named Masstree [41]. It is well-known for its multi-core scalability and high efficiency. To support range searching, a global Masstree instance is created at startup and shared by all the server cores. The keys, versions and pointers are stored at leaf nodes. Similar to FlatStore-H, the clients still choose a specific server core according to the hash value of the key to post their requests, so as to reduce the chance of conflicts when updating the Masstree.

4.3 RDMA-based FlatRPC

RDMA is capable of directly accessing the remote memory without the involvements of remote CPUs. It provides two types of verbs: (1) *Two-sided*, such as RDMA `send/recv`. (2) *One-sided*, such as RDMA `read/write`. RDMA requests are sent over Queue Pairs (QPs). Similar to the FaRM RPC [23],

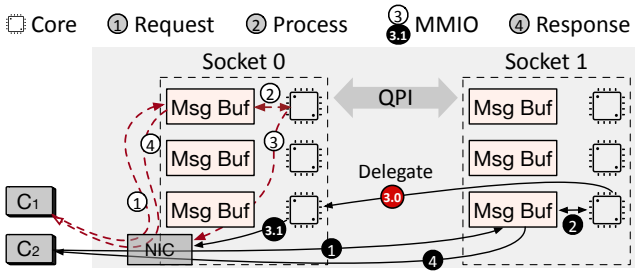

Figure 6. Data Flow of FlatRPC. C_1 sends a request to its agent core (red dashed line) and C_2 sends to a normal core (black solid line).

we use RDMA `write` verb to send both request and response messages between the clients and servers. As described above, FlatStore requires that a client can send requests to server cores with IDs specified by this client, indicating that each server node has to create $N_t \times N_c$ of queue pairs (where N_t is the number of cores per server and N_c is the number of clients). This can lead to severe scalability issue when the number of clients increases, since the NIC has limited cache space [17, 23, 32, 33].

To support efficient message passing, we introduce the *RDMA-based FlatRPC*, which lets the clients directly write requests to the message buffer of a server core, but reply messages are delegated to the agent core to send back. When a client connects to each server node, only one QP is created between it and an *agent core* in the server. The agent core is randomly chosen from the socket close to the NIC. Each server core also pre-allocates a *message buffer* for this client to store the incoming messages, but sharing a single QP. As shown in Figure 6, to post a message, the client writes the message payload directly to the message buffer of a specific core (step 1). Each server core polls the message buffer to extract the new message and process it (step 2). After this, a response message is prepared and posted according to the following rule: (1) If this core happens to be the agent core, it directly posts a `write` verb with Memory Mapped I/O (MMIO) (step 3 of the red dashed line). (2) Otherwise, it delegates the verb command to the agent core through shared memory (step 3.0 and 3.1). When the NIC receives the verb, it fetches the payload and sends it to the client (step 4).

FlatRPC reduces the number of QPs to N_c, with the extra cost of the inter-core delegation. However, we find that it's worth the effort because (1) the verb commands are light-weight (several bytes). Besides, the agent core can use *prefetch* instructions to accelerate the delegation; (2) the delegate phase gathers all the verb commands to the socket close to the NIC, further reducing the MMIO overhead. On our platform (in Section 5), FlatRPC provides throughput of 52.7 Mop/s with one FlatRPC server and 12 client nodes (12×24 client threads), which is 1.5× higher than that of "all-to-all connected" approach.

RDMA offloading. Some works suggest to offload the processing of *Get* to the clients with one-sided RDMA verbs (e.g., RDMA `read`) [42, 43, 58], so as to relieve the burden of server CPUs. In our implemented prototype of FlatStore, however, we observe that the overall throughput based on RDMA `read` is respectively 57% and 21% slower than the two-sided approach (i.e., RPC) for 100%-*Get* and 50%-*Get* workloads. This is because we need to post multiple RDMA `reads` to locate the KV items remotely. Our experimental results are also consistent with HERD's [31]. Therefore, in FlatStore, both *Get* and *Put* requests are processed at server-side through RPC primitives.

5 Evaluation

In this section, we evaluate the performance of FlatStore and other existing KV stores. We also conduct experiments to analyze the benefits of each mechanism design.

Hardware Platform. Our cluster consists of one server node and 12 client nodes. The server node is equipped with four Optane DCPMM (256 GB per DIMM, 1 TB in total), 128 GB of DRAM memory and two 2.6GHz Intel Xeon Gold 6240M CPUs (36 cores in total), installed with Ubuntu 18.04. Each client node has 128 GB of DRAM memory and two 2.2GHz Intel Xeon E5-2650 v4 CPUs (24 cores in total), installed with CentOS 7.4. All these servers are connected with a Mellanox MSB7790-ES2F switch using MCX555A-ECAT ConnectX-5 EDR HCAs, which support 100Gbps Infiniband network. To explore the peak throughput that each system can achieve, we let the clients post multiple requests asynchronously and poll the completion in a batch manner [32, 33] (default client batchsize is 8).

Compared Systems. Four state-of-the-art persistent index schemes are chosen to compare with (see Table 1). 1) *Hash-based.* Level-Hashing [64] and CCEH [44] are used to compare with FlatStore-H. We directly use their default settings. In our implementation, clients directly route their requests to a specific core according to the key hash, so we create a Level-Hashing/CCEH instance for each server core and remove all the lock operations inside them. As the hash table resizing impacts their performance, the hash table is created with big enough size and we collect the performance before they are resized. 2) *Tree-based.* We use FPTree [47] and FAST-FAIR [27] to compare with FlatStore-M. FPTree is not open-sourced so we implement it based on STX B$^+$-Tree[1]. Similar to FlatStore-M, a single FPTree/FAST-FAIR instance is shared by all the server cores to support range search. All the compared index schemes store the KV records with our proposed *Lazy-persist allocator*, while only storing a pointer in the index to point to the actual data.

Table 1. Description of Compared Index Schemes.

Type	Name	Description
Hash	CCEH	Three level (directory, segments, buckets), 4 slots in a bucket
	Level-Hashing	Two-level (top/bottom level), 4 slots in a bucket
Tree	FPTree	Inner nodes are placed in DRAM.
	FAST&FAIR	All nodes are placed in PM.

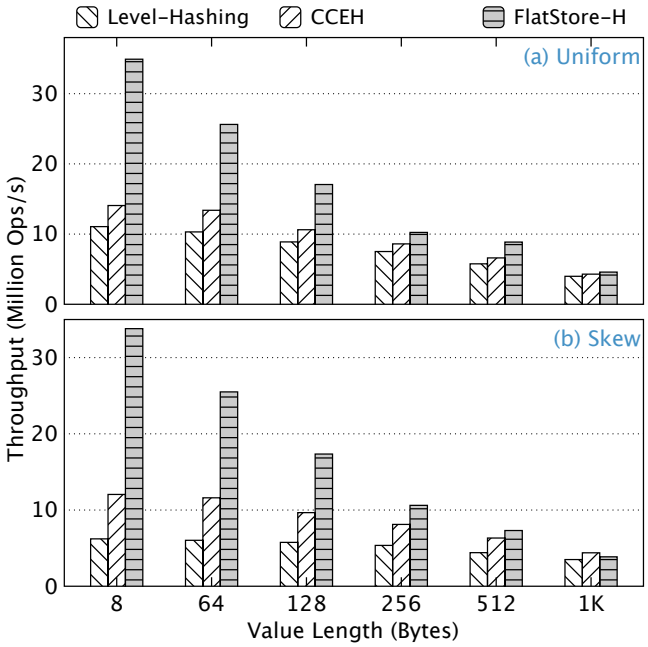

Figure 7. Put Performance of FlatStore-H.

5.1 YCSB - Microbenchmark

We first evaluate the *Put* performance of FlatStore by varying the item size and skewness with YCSB workloads [19]. All of them have the key range of 192 million and key size of 8 bytes. The skew workload uses a zipfan distribution of key popularity (skewness of 0.99, the default setting in YCSB). FlatStore targets as improving the performance of write-intensive and small-sized workloads, so we only show the result of *Put* operation in this section, more detailed evaluation with higher *Get* ratio is given in Section 5.2.

FlatStore-H. The experimental results are shown in Figure 7 and we make the following observations:

First, FlatStore-H shows higher throughput than Level-Hashing and CCEH and such performance advantage is especially obvious for small-sized workloads. For 8-byte values, FlatStore-H achieves 35 Mops/s for uniform workloads, 2.5 times and 3.2 times the throughput of CCEH and Level-Hashing; and 34 Mops/s for skew workloads, 2.8 times and 5.4 times higher than the other two systems. FlatStore-H's high efficiency comes from the following aspects: 1) By introducing the compacted OpLog, FlatStore-H reduces the

[1] https://panthema.net/2007/stx-btree

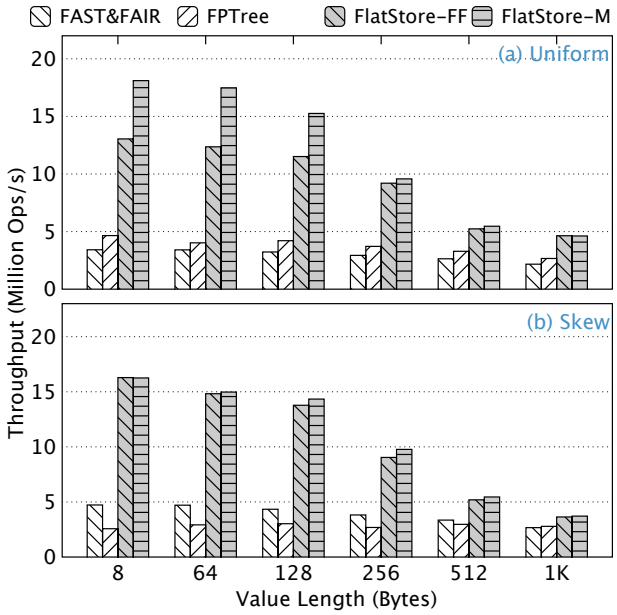

Figure 8. Put Performance of FlatStore-M.

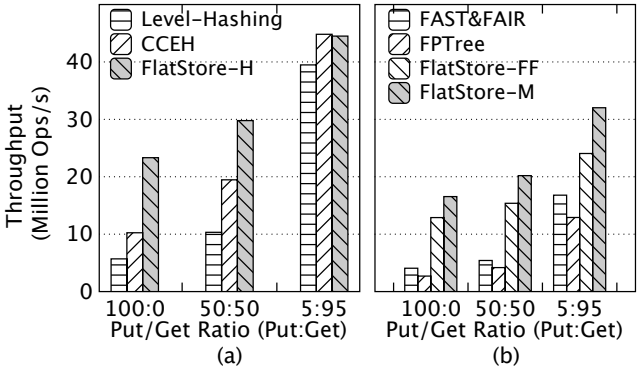

Figure 9. Throughput with Facebook ETC.

number of persisting operations for each *Put* operations by simply appending a log entry. Instead, both CCEH and Level-Hashing need to flush the index entry, bitmap in the bucket, and the actual KV record in the allocator for each update. 2) When the two keys conflict, Level-Hashing needs to re-hash the related entries, and CCEH also needs to split the segments when they are full, which amplifies the flushing times. 3) FlatStore-H efficiently merges the small-sized log entries from multiple cores with pipelined HB, and persists them in a batch manner, which further reduces the number of writes. The performance of CCEH is slightly higher than Level-Hashing, since it adopts different collision policy, reducing the chance of conflicts. All the three systems show very close performance for large KVs since they are mainly restricted by the PM bandwidth.

Second, FlatStore-H achieves higher performance improvement with skew workloads. Both CCEH and Level-Hashing "in-place" update the index structure (e.g., the bitmap, 8-byte pointers, etc.), so the chance to repeatedly flush the same cacheline is increased with skew workloads, which impacts their performance to some extent. Besides, we observe that the "request stealing" policy in pipelined HB has its nature to better support skew workloads: those non-busy cores have higher opportunity to become the *leader*, and help the busy cores to flush the log entries. In CCEH and level-Hashing, the requests are strictly partitioned to different cores so the busiest core restricts the overall performance. We also notice that FlatStore-H loses such performance advantages with large KVs. This is because the large KVs are still persisted locally by each server core (similar to CCEH and Level-Hashing).

FlatStore-M. To exclude the performance advantages brought by Masstree, we also implement FlatStore-FF by replacing Masstree with FAST&FAIR. Similarly, FAST&FAIR is placed in DRAM to work as the volatile index structure. The experimental results are shown in Figure 8 and we observe that: 1) The performance improvements brought by FlatStore-M is more significant than FlatStore-H. For 8-byte records, FlatStore-M can perform 18 Mops/s with the uniform workload, 3.4 times and 4.5 times the throughput of FPTree and FAST&FAIR. For skew workload, the throughput of FlatStore-M is 16 Mops/s, 6.3 times and 3.4 times higher than FPTree and FAST&FAIR. This is mainly because the tree nodes are frequently merged and split in these persistent indexes, which is more likely to cause write amplification. 2) FlatStore-M has higher throughput than FlatStore-FF, especially for the uniform workload. Masstree is designed to improve the multicore scalability of B^+-Tree in DRAM. Therefore, it is more efficient than the volatile version of FAST&FAIR. Even so, the throughput of FlatStore-FF is still far higher than FPTree and FAST&FAIR for all the evaluated workloads. 3) FPTree shows higher throughput than FAST&FAIR with uniform workloads. This is because FPTree only places its leaf nodes in PM and other inner nodes are directly placed in DRAM. Instead, all the tree nodes in FAST&FAIR are placed in NVM, causing much more persistence overhead. Even so, FAST&FAIR has higher performance with skew workload since it incorporates more fine-grained lock design and causes less contention.

5.2 Facebook ETC Pool - Production Workload

We emulate the ETC pool at Facebook [51] as the production workload to evaluate the behavior of FlatStore. It has a trimodal item distribution, where the item can be tiny (1-13 bytes), small (14-300 bytes) or large (larger than 300 bytes). Out of the key space (192 million), 40% correspond to tiny items, 55% correspond to small items and the remaining 5% to large ones. We use zipfian distribution (0.99) on the sets of tiny and small items, which is representative of the strong skew of many production workloads. Large items, instead,

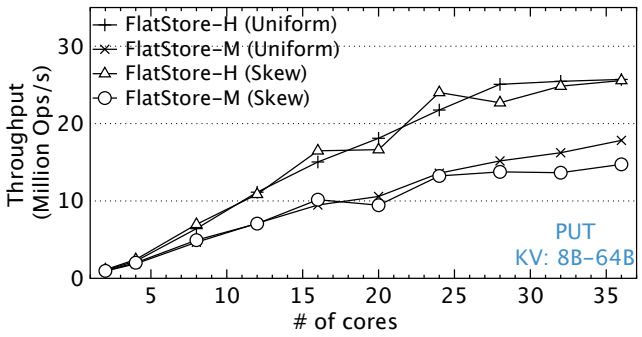

Figure 10. Throughput with Varying Server Cores.

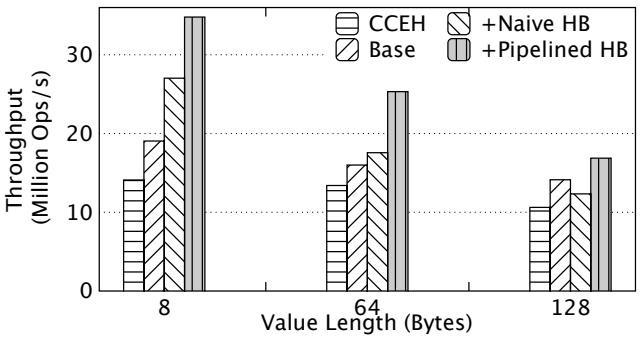

Figure 11. Benefits of Each Optimizations in FlatStore.

are much fewer and exhibit much higher variability, and are chosen uniformly at random [22]. ETC pool is originally read-dominated, but we consider both read-dominated and write-intensive workloads, corresponding, respectively, to a 95:5 and 50:50 *Get*:*Put* ratio [22]. For reference, the results of 100%-*Put* are given too. The evaluation results are shown in Figure 9 and we make the following observations:

(1) Both FlatStore-H and FlatStore-M show significant higher throughput than the compared systems, and the performance advantage over tree-based ones is more obvious. This is also consistent with the previous evaluation results in Section 5.1. Specifically, FlatStore-H achieves 23 Mops/s with 100:0 workloads, 2.3 and 4.1 times the throughput of Level-Hashing and CCEH. FlatStore-M improves performance by 4× and 6× compared to that of FAST&FAIR and FPTree. We explain the high performance of FlatStore from the following aspects: 1) A large proportion of KVs in ETC pool are small-sized (e.g., 95% are smaller than 300 bytes), and FlatStore is good at handling such small-sized updates by merging them into a larger one with batching. 2) FlatStore avoids updating pointers as in the index structure, since it simply appends log entries in the log. As a result, FlatStore greatly reduces the number of PM writes.

(2) FlatStore mainly focuses on improving the write performance, so FlatStore-H shows almost the same performance as that of CCEH and Level-Hashing as the read/write ratio increases. We notice that FlatStore-M still has higher performance with 5:95 workload. This is because, in a tree-based index, the *Put* operations still account for a non-negligible part of overhead, even if it has a small proportion.

5.3 Multicore Scalability

In this section, we vary the number of server cores to reveal the multicore scalability of FlatStore. In our evaluation, the cores are evenly distributed to each of the NUMA domain and the group size of FlatStore increases along with the number of the allocated cores in each socket. We use both uniform and skewed workloads (100%-*Put* and 64-byte KV records) and the experimental results are shown in Figure 10. We can observe that FlatStore-H and FlatStore-M scales almost linearly as the number of server cores increase to 26 for both uniform and skewed workloads. In FlatStore-H, both the in-memory hash table and the NVM space are privately managed by each server core, so there is no synchronization overhead between the cores. Besides, pipelined HB also helps to rebalance the loads between different server cores, which enables FlatStore-H to exhibit scalable performance even with skew workloads. Similarly, by putting the scalable volatile index (i.e., Masstree) and the efficient NVM storage engine (i.e., FlatStore) altogether, FlatStore-M is also highly scalable. When there are more threads, all of their performance improvements slow down, which is mainly limited by the PM bandwidth.

5.4 Analysis of Each Optimization

In this section, we analyze the performance improvements brought each optimization (due to the limited space, we only evaluate FlatStore-H).

First, we measure the benefit of compacted OpLog. To achieve this, we implement a Base version, which keeps the log-structured design but lets each core process one request at a time (pipelined HB is disabled). Figure 11 (a) shows their *Put* throughput with varying value sizes. We observe that *Base* outperforms CCEH by 29.3% on average, because CCEH needs to perform multiple updates in its index structure and the actual KV records, while the log-structured design in *Base* reduce the number of persistence operations.

Second, we measure the benefit of pipelined HB by implementing a a *naive HB* version, which steals requests from other cores, but releases the lock only after the log entries have been persisted. *Naive HB* shows higher performance than *Base* for small KVs, but underperforms it with 128 B values. This is because *Naive HB* pays less overhead to flush 8 B or 64 B values, without delaying other server cores too much. Pipelined HB brings significant benefit in all the cases because it 1) merges both the small-sized log entries and small KVs into fewer cachelines and greatly reduces the number of PM writes; 2) releases the lock in advance, so that other server cores are enabled to execute the next batch in parallel.

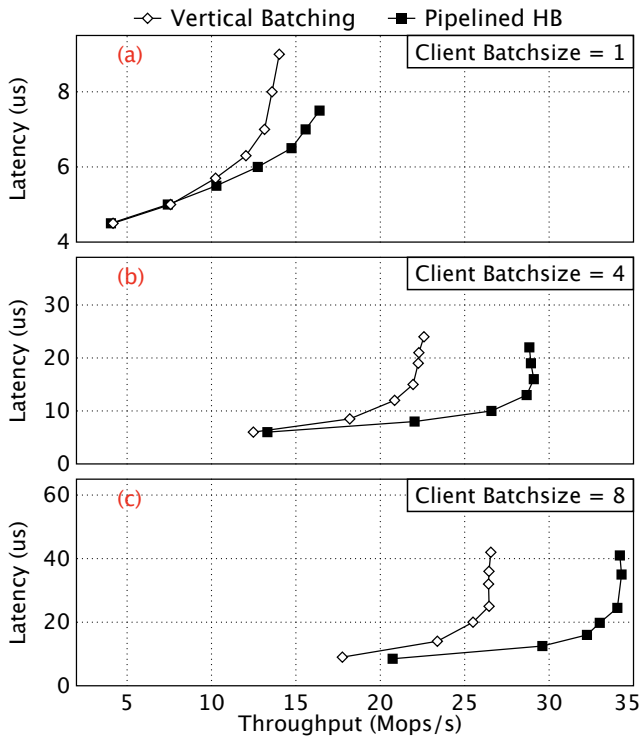

Figure 12. Performance Comparison Between Pipelined HB and Vertical Batching.

We then measure the impact of Pipelined HB on latency. To lower latency, Pipelined HB tries to build a batch faster by letting one core steal the log entries from its sibling cores. To understand the effects of such design, we also implement *Vertical Batching*, which lets each core scan its own message buffers from the beginning to the end and build a batch with the discovered new messages. Actually, *Vertical Batching* is equivalent to Pipelined HB when its group size is set to 1. We compare the throughput and latency of both *Vertical Batching* and Pipelined HB for a varying number of client nodes and client batch size. We use uniform workloads with *Put* operations. The latencies of *Vertical Batching* and Pipelined HB are collected when the completion message is returned to the clients. The results are shown in Figure 12 and we make the following observations:

1) Pipelined HB exhibits comparable or higher throughput and lower latency when there are less batching opportunities. As shown in Figure 12 (a), the client batch size is set to 1. Pipelined HB shows the same throughput and latency as Vertical Batching when the number of clients is 1 or 2, but it soon outperforms *Vertical Batching* in terms of both throughput and latency when there are more client nodes. We also notice that their performance gap is further widening as the number of clients increases. Since the requests posted by each client is infrequent, it's more difficult for the server cores in *Vertical Batching* to accumulate enough new requests. However, Pipelined HB accumulates the requests from other cores, so the cores can easily build a batch with large size and persist them in parallel.
2) When there are sufficient batch opportunities (as shown in Figure 12(b) and (c)), Pipelined HB significantly outperforms Vertical batching in throughput, while still achieves comparable latency. Specifically, with client batch size of 8, the throughput of Pipelined HB is 23% higher than that of Vertical batching.

In summary, Pipelined HB is capable of providing both high throughput and low latency. It is also gifted in adapting to the clients' requests with varying concurrency, and such property of flexibility is crucial in real-world systems.

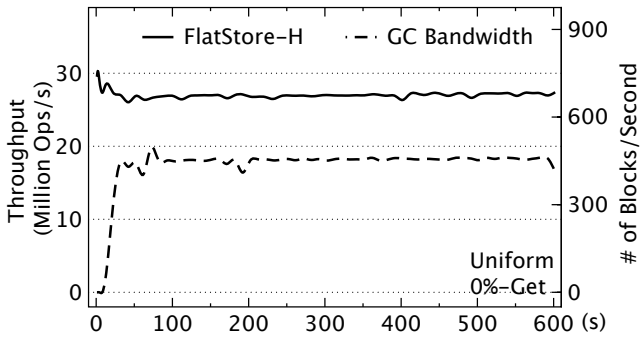

Figure 13. Efficiency of Garbage Collection.

5.5 Garbage Collection Overhead

To evaluate the efficiency of log cleaning in FlatStore, we run FlatStore-H with ETC workload (50% *Get*) and with a duration of 10 minutes. As shown in Figure 13, the throughput of FlatStore-H drops slightly from 29.8 to 27 Mops/s (with the reduction of only 10%) when the GC process is triggered. After that, both the throughput of FlatStore-H and the log cleaning rate remain stable for the rest time. Several reasons account for the efficient garbage collection: (1) The background cleaner reclaims the blocks without blocking the normal requests posted by the clients; (2) The OpLog is light-weight since it only keeps the index metadata and small KVs so the copying overhead is low; and (3) Each GC group is equipped with a cleaner to reclaim the obsolete log entries in parallel.

6 Related Works

NVM-based KV stores. Existing NVM-based KV stores can be roughly divided into two types: (1) Hash-based. HiKV [59] constructs a hybrid index to support rich key-value operations efficiently. It builds a persistent hash table for fast index searching and a volatile B$^+$-Tree for range scan. Bullet [26] tries to close the performance gap between volatile and persistent KV stores using a technique called cross-referencing logs (CRLs) to keep most NVM updates off the critical path. (2) Tree-based. A number of previous works [15, 47, 57, 61]

are devoted to reducing the persistency overhead on persistent B$^+$-Tree or its variants. Both NV-Tree [61] and FPTree [47] place the inner nodes in DRAM and the leaf nodes in NVM, so as to reduce the flush operations. wB$^+$Tree [15], NV-Tree [61] and FPTree [47] keep the entries in each leaf node unsorted, targeting at dismissing the data shifting overhead in NVM. To consistently manage the NVM space and avoid memory leak, Both Bullet [26] and FPTree [47] adopts persistent reference pointers (a.k.a., *root pointers*) to track the used data blocks. However, CDDS-Tree [57], wB$^+$Tree [15], NV-Tree [61] and HiKV [59] just ignore this issue, with the risk of causing memory leak.

Log-structured Persistent Memory. LSNVMM [25] is a log-structured transactional system for NVM. It updates the data by appending it to the end of the log, so as to reduce NVM space fragmentation and avoid redo/undo logging overhead as in traditional transactional system. NOVA [60] is a highly scalable persistent memory file system. It improves multi-core scalability by organizing each inode with an independent log structure. For crash consistency, NOVA uses copy-on-write technique to modify the file data and introduces light-weight journal for complicated operations involving updates to multiple logs (e.g., `rename`). NOVA also relies on the log entry to recover its allocation metadata. However, it's less challenging for a file system to achieve this since all its data pages are 4KB-aligned. More than simply adopting the log-structured layout, FlatStore steps further to alleviates the persistency overhead by exploring the benefits of batching.

Batch Processing. Batch processing is traditionally applied to accelerate the *network transferring* and *data storage* respectively. In terms of networking, distributed systems usually delay the data transferring to accumulate multiple remote requests and send them in a single batch to reduce the network round-trips (e.g., the NameNode in HDFS [56]). Besides, batch processing systems like MapReduce [20], Spark [62], and Apache Flink [13] organize the data records into static data sets and process them in a time-agnostic fashion. Such coarse-grained management greatly reduces the network interaction frequency. For data storage, as traditional hard disks perform badly for random accesses, system software, such as journaling file systems (e.g., EXT4 [1], XFS [28], ReiserFS [4]) and LSM-Tree-based databases (e.g, LevelDB [2]) often buffers the written data in DRAM, and asynchronously flush it to the disk sequentially at checkpoint in a batch manner. Different from the above batching systems, FlatStore profits from batching without delaying the clients. Moreover, since NVM has different flush granularity, FlatStore only batches the small-sized metadata and KV items in the log.

RDMA and Persistent Memory. Persistent memory has been well studied in file systems [16, 18, 24, 60], key-value stores and data structures in the past decade. Recently, it becomes popular to build efficient distributed storage systems with persistent memory and RDMA network. Octopus [39] is a distributed file system by abstracting a shared persistent memory pool to reduce redundant memory copies in data transferring. Hotpot [55] manages the PMs from different nodes into a global address space, and supports fault-tolerance via data replication. AsymNVM [40] is a generic framework for asymmetric disaggregated non-volatile memories. It implements the fundamental primitives for building remote data structures, including space management, concurrency control, crash consistency and replication.

7 Conclusion

Existing KV stores generate a huge number of small-sized writes when processing today's popular workloads, which mismatches with the persistence granularity in PMs. To address this issue, we decouple the KV store into a volatile index and log-structured storage, and thus propose a PM-based KV storage engine named FlatStore. It manages the index and small-sized KVs with a per-core OpLog, so as to better support batching. *Pipelined horizontal batching* is also proposed to provide high-throughput and low-latency request processing. Evaluations show that FlatStore significantly outperforms existing KV stores.

8 Acknowledgements

We sincerely thank our shepherd Yu Hua for helping us improve the paper. We also thank the anonymous reviewers for their feedback and suggestions. This work is supported by the National Key Research & Development Program of China (Grant No. 2018YFB1003301), the National Natural Science Foundation of China (Grant No. 61832011, 61772300), the Research and Development Plan in Key field of Guangdong Province (Grant No. 2018B010109002), and the Project of ZTE (Grant No. 20182002008).

References

[1] [n.d.]. Ext4. https://ext4.wiki.kernel.org/.
[2] [n.d.]. LevelDB, A fast and lightweight key/value database library by Google. http://code.google.com/p/leveldb/.
[3] [n.d.]. The Persistent Memory Development Kit. "pmem.io".
[4] [n.d.]. ReiserFS. http://reiser4.wiki.kernel.org.
[5] 2017. Product Brief of ConnectX-5 VPI Card. "http://www.mellanox.com/related-docs/prod_adapter_cards/PB_ConnectX-5_VPI_Card.pdf".
[6] 2018. Intel's Apache Pass. "https://software.intel.com/en-us/articles/intel-optane-dc-persistent-memory-for-greater-capacity-affordability-persistence".
[7] Amr Ahmed, Moahmed Aly, Joseph Gonzalez, Shravan Narayanamurthy, and Alexander J. Smola. 2012. Scalable Inference in Latent Variable Models. In *Proceedings of the Fifth ACM International Conference on Web Search and Data Mining (WSDM '12)*. ACM, New York, NY, USA, 123–132. https://doi.org/10.1145/2124295.2124312
[8] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. 2012. Workload analysis of a large-scale key-value store. In *ACM SIGMETRICS Performance Evaluation Review*, Vol. 40. ACM, 53–64.
[9] IG Baek, MS Lee, S Seo, MJ Lee, DH Seo, D-S Suh, JC Park, SO Park, HS Kim, IK Yoo, et al. 2004. Highly scalable nonvolatile resistive memory

using simple binary oxide driven by asymmetric unipolar voltage pulses. In *Electron Devices Meeting, 2004. IEDM Technical Digest. IEEE International*. IEEE, 587–590.

[10] Emery D Berger, Kathryn S McKinley, Robert D Blumofe, and Paul R Wilson. 2000. Hoard: A scalable memory allocator for multithreaded applications. In *ACM SIGARCH Computer Architecture News*, Vol. 28. ACM, 117–128.

[11] Kumud Bhandari, Dhruva R Chakrabarti, and Hans-J Boehm. 2016. Makalu: Fast recoverable allocation of non-volatile memory. In *ACM SIGPLAN Notices*, Vol. 51. ACM, 677–694.

[12] Silas Boyd-Wickizer, M Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. 2014. OpLog: a library for scaling update-heavy data structures. (2014).

[13] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. 2015. Apache Flink: Stream and Batch Processing in a Single Engine. *IEEE Data Eng. Bull.* 38 (2015), 28–38.

[14] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. 2008. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)* 26, 2 (2008), 4.

[15] Shimin Chen and Qin Jin. 2015. Persistent b+-trees in non-volatile main memory. *Proceedings of the VLDB Endowment* 8, 7 (2015), 786–797.

[16] Y. Chen, Y. Lu, P. Chen, and J. Shu. 2019. Efficient and Consistent NVMM Cache for SSD-Based File System. *IEEE Trans. Comput.* 68, 8 (Aug 2019), 1147–1158. https://doi.org/10.1109/TC.2018.2870137

[17] Youmin Chen, Youyou Lu, and Jiwu Shu. 2019. Scalable RDMA RPC on Reliable Connection with Efficient Resource Sharing. In *Proceedings of the Fourteenth EuroSys Conference 2019 (EuroSys '19)*. ACM, New York, NY, USA, Article 19, 14 pages. https://doi.org/10.1145/3302424.3303968

[18] Youmin Chen, Jiwu Shu, Jiaxin Ou, and Youyou Lu. 2018. HiNFS: A Persistent Memory File System with Both Buffering and Direct-Access. *ACM Trans. Storage* 14, 1, Article 4 (April 2018), 30 pages. https://doi.org/10.1145/3204454

[19] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC)*. ACM, New York, NY, USA, 143–154.

[20] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: simplified data processing on large clusters. *Commun. ACM* 51, 1 (2008), 107–113.

[21] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. 2007. Dynamo: amazon's highly available key-value store. In *ACM SIGOPS operating systems review*, Vol. 41. ACM, 205–220.

[22] Diego Didona and Willy Zwaenepoel. 2019. Size-aware Sharding For Improving Tail Latencies in In-memory Key-value Stores.. In *NSDI*. 79–94.

[23] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. 2014. FaRM: fast remote memory. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. 401–414.

[24] Subramanya R. Dulloor, Sanjay Kumar, Anil Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff Jackson. 2014. System Software for Persistent Memory. In *Proceedings of the Ninth European Conference on Computer Systems (EuroSys)*. ACM, New York, NY, USA, Article 15, 15 pages.

[25] Qingda Hu, Jinglei Ren, Anirudh Badam, and Thomas Moscibroda. 2017. Log-structured non-volatile main memory. In *Proceedings of 2017 USENIX Annual Technical Conference (USENIX ATC)*.

[26] Yihe Huang, Matej Pavlovic, Virendra Marathe, Margo Seltzer, Tim Harris, and Steve Byan. 2018. Closing the Performance Gap Between Volatile and Persistent Key-Value Stores Using Cross-Referencing Logs. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, 967–979.

[27] Deukyeon Hwang, Wook-Hee Kim, Youjip Won, and Beomseok Nam. 2018. Endurable Transient Inconsistency in Byte-addressable Persistent B+-tree. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies (FAST'18)*. USENIX Association, Berkeley, CA, USA, 187–200. http://dl.acm.org/citation.cfm?id=3189759.3189777

[28] Silicon Graphics Inc. [n.d.]. XFS User Guide: A guide for XFS filesystem users and administrators. http://xfs.org/index.php/XFS_Papers_and_Documentation.

[29] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R Dulloor, et al. 2019. Basic performance measurements of the intel optane DC persistent memory module. *arXiv preprint arXiv:1903.05714* (2019).

[30] Anuj Kalia, Michael Kaminsky, and David Andersen. 2019. Datacenter RPCs can be general and fast. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. 1–16.

[31] Anuj Kalia, Michael Kaminsky, and David G Andersen. 2015. Using RDMA efficiently for key-value services. *ACM SIGCOMM Computer Communication Review* 44, 4 (2015), 295–306.

[32] Anuj Kalia, Michael Kaminsky, and David G Andersen. 2016. Design Guidelines for High Performance RDMA Systems. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*.

[33] Anuj Kalia, Michael Kaminsky, and David G Andersen. 2016. FaSST: fast, scalable and simple distributed transactions with two-sided RDMA datagram RPCs. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, 185–201.

[34] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. 2018. Pocket: Elastic Ephemeral Storage for Serverless Analytics. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI'18)*. USENIX Association, Berkeley, CA, USA, 427–444. http://dl.acm.org/citation.cfm?id=3291168.3291200

[35] Benjamin C. Lee, Engin Ipek, Onur Mutlu, and Doug Burger. 2009. Architecting phase change memory as a scalable dram alternative. In *Proceedings of the 36th annual International Symposium on Computer Architecture (ISCA)*. ACM, New York, NY, USA, 2–13.

[36] Changman Lee, Dongho Sim, Jooyoung Hwang, and Sangyeun Cho. 2015. F2FS: A New File System for Flash Storage. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST)*. USENIX, Santa Clara, CA. https://www.usenix.org/conference/fast15/technical-sessions/presentation/lee

[37] Bojie Li, Zhenyuan Ruan, Wencong Xiao, Yuanwei Lu, Yongqiang Xiong, Andrew Putnam, Enhong Chen, and Lintao Zhang. 2017. KV-Direct: High-Performance In-Memory Key-Value Store with Programmable NIC. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17)*. ACM, New York, NY, USA, 137–152. https://doi.org/10.1145/3132747.3132756

[38] Mu Li, David G Andersen, Jun Woo Park, Alexander J Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J Shekita, and Bor-Yiing Su. 2014. Scaling Distributed Machine Learning with the Parameter Server.. In *OSDI*, Vol. 14. 583–598.

[39] Youyou Lu, Jiwu Shu, Youmin Chen, and Tao Li. 2017. Octopus: an RDMA-enabled distributed persistent memory file system. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. USENIX Association, 773–785.

[40] Teng Ma, Mingxing Zhang, Kang Chen, Xuehai Qian, Zhuo Song, and Yongwei Wu. 2020. AsymNVM: An Efficient Framework for Implementing Persistent Data Structures on Asymmetric NVM Architecture. In *the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM.

[41] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. 2012. Cache craftiness for fast multicore key-value storage. In *Proceedings of the*

7th ACM european conference on Computer Systems. ACM, 183–196.

[42] Christopher Mitchell, Yifeng Geng, and Jinyang Li. 2013. Using One-Sided RDMA Reads to Build a Fast, CPU-Efficient Key-Value Store. In *Presented as part of the 2013 USENIX Annual Technical Conference (USENIX ATC 13)*. 103–114.

[43] Christopher Mitchell Kate Montgomery, Lamont Nelson, Siddhartha Sen, and Jinyang Li. 2016. Balancing CPU and network in the cell distributed B-Tree store. In *2016 USENIX Annual Technical Conference*. 451.

[44] Moohyeon Nam, Hokeun Cha, Young-ri Choi, Sam H Noh, and Beomseok Nam. 2019. Write-optimized dynamic hashing for persistent memory. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*. 31–44.

[45] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, et al. 2013. Scaling Memcache at Facebook.. In *nsdi*, Vol. 13. 385–398.

[46] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. 1996. The log-structured merge-tree (LSM-tree). *Acta Informatica* 33, 4 (1996), 351–385.

[47] Ismail Oukid, Johan Lasperas, Anisoara Nica, Thomas Willhalm, and Wolfgang Lehner. 2016. FPTree: A hybrid SCM-DRAM persistent and concurrent B-tree for storage class memory. In *Proceedings of the 2016 International Conference on Management of Data*. ACM, 371–386.

[48] John K. Ousterhout, Arjun Gopalan, Ashish Gupta, Ankita Kejriwal, Collin Lee, Behnam Montazeri, Diego Ongaro, Seo Jin Park, Henry Qin, Mendel Rosenblum, Stephen M. Rumble, Ryan Stutsman, and Stephen Yang. 2015. The RAMCloud Storage System. *ACM Trans. Comput. Syst.* 33 (2015), 7:1–7:55.

[49] Henry Qin, Qian Li, Jacqueline Speiser, Peter Kraft, and John Ousterhout. 2018. Arachne: Core-aware Thread Management. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI'18)*. USENIX Association, Berkeley, CA, USA, 145–160. http://dl.acm.org/citation.cfm?id=3291168.3291180

[50] Moinuddin K. Qureshi, Vijayalakshmi Srinivasan, and Jude A. Rivers. 2009. Scalable high performance main memory system using phase-change memory technology. In *Proceedings of the 36th annual International Symposium on Computer Architecture (ISCA)*. ACM, New York, NY, USA, 24–33.

[51] Drew S Roselli, Jacob R Lorch, Thomas E Anderson, et al. 2000. A Comparison of File System Workloads.. In *Proceedings of 2000 USENIX Annual Technical Conference*. USENIX, Berkeley, CA, 41–54.

[52] Mendel Rosenblum and John K Ousterhout. 1992. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems (TOCS)* 10, 1 (1992), 26–52.

[53] Stephen M Rumble, Ankita Kejriwal, and John K Ousterhout. 2014. Log-structured memory for DRAM-based storage.. In *FAST*, Vol. 14. 1–16.

[54] Stephen M Rumble, Diego Ongaro, Ryan Stutsman, Mendel Rosenblum, and John K Ousterhout. 2011. It's Time for Low Latency.. In *HotOS*, Vol. 13. 11–11.

[55] Yizhou Shan, Shin-Yeh Tsai, and Yiying Zhang. 2017. Distributed shared persistent memory. In *Proceedings of the 2017 Symposium on Cloud Computing*. ACM, 323–337.

[56] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. 2010. The Hadoop Distributed File System. *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)* (2010), 1–10.

[57] Shivaram Venkataraman, Niraj Tolia, Parthasarathy Ranganathan, Roy H Campbell, et al. 2011. Consistent and Durable Data Structures for Non-Volatile Byte-Addressable Memory.. In *Proceedings of the 9th USENIX Conference on File and Storage Technologies (FAST)*. USENIX, Berkeley, CA, 61–75.

[58] Yandong Wang, Li Zhang, Jian Tan, Min Li, Yuqing Gao, Xavier Guerin, Xiaoqiao Meng, and Shicong Meng. 2015. HydraDB: a resilient RDMA-driven key-value middleware for in-memory cluster computing. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 22.

[59] Fei Xia, Dejun Jiang, Jin Xiong, and Ninghui Sun. 2017. Hikv: A hybrid index key-value store for dram-nvm memory systems. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. 349–362.

[60] Jian Xu and Steven Swanson. 2016. NOVA: a log-structured file system for hybrid volatile/non-volatile main memories. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*. 323–338.

[61] Jun Yang, Qingsong Wei, Cheng Chen, Chundong Wang, Khai Leong Yong, and Bingsheng He. 2015. NV-Tree: Reducing Consistency Cost for NVM-based Single Level Systems.. In *FAST*, Vol. 15. 167–181.

[62] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: Cluster computing with working sets. *HotCloud* 10, 10-10 (2010), 95.

[63] Ping Zhou, Bo Zhao, Jun Yang, and Youtao Zhang. 2009. A durable and energy efficient main memory using phase change memory technology. In *Proceedings of the 36th annual International Symposium on Computer Architecture (ISCA)*. ACM, New York, NY, USA, 14–23.

[64] Pengfei Zuo, Yu Hua, and Jie Wu. 2018. Write-optimized and High-performance Hashing Index Scheme for Persistent Memory. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI'18)*. USENIX Association, Berkeley, CA, USA, 461–476. http://dl.acm.org/citation.cfm?id=3291168.3291202