Viper: An Efficient Hybrid PMem-DRAM Key-Value Store

Lawrence Benson, Hendrik Makait, Tilmann Rabl
{firstname.lastname}@hpi.de
Hasso Plattner Institute, University of Potsdam

ABSTRACT

Key-value stores (KVSs) have found wide application in modern software systems. For persistence, their data resides in slow secondary storage, which requires KVSs to employ various techniques to increase their read and write performance from and to the underlying medium. Emerging persistent memory (PMem) technologies offer data persistence at close-to-DRAM speed, making them a promising alternative to classical disk-based storage. However, simply drop-in replacing existing storage with PMem does not yield good results, as block-based access behaves differently in PMem than on disk and ignores PMem's byte addressability, layout, and unique performance characteristics. In this paper, we propose three PMem-specific access patterns and implement them in a hybrid PMem-DRAM KVS called *Viper*. We employ a DRAM-based hash index and a PMem-aware storage layout to utilize the random-write speed of DRAM and efficient sequential-write performance PMem. Our evaluation shows that Viper significantly outperforms existing KVSs for core KVS operations while providing full data persistence. Moreover, Viper outperforms existing PMem-only, hybrid, and disk-based KVSs by 4–18x for write workloads, while matching or surpassing their *get* performance.

PVLDB Reference Format:
Lawrence Benson, Hendrik Makait, Tilmann Rabl. Viper: An Efficient Hybrid PMem-DRAM Key-Value Store. PVLDB, 14(9): 1544 - 1556, 2021.
doi:10.14778/3461535.3461543

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/hpides/viper.

1 INTRODUCTION

Persistent key-value stores (KVSs) have become a widely used alternative type of data store next to classical relational database management systems (RDBMSs). Different to RDBMSs, KVSs store schema-less data (*value*) retrievable through a given *key*. KVS workloads also differ from classical RDBMS workloads in that they are write-heavy and nearly exclusively operate on single records [24]. These workload characteristics allow for a variety of KVS applications, ranging from storage engines in SQL systems [10], over state-storage for stream processing engines [4, 56], to caches for web applications [45]. On a large scale, these use-cases all require high performance and strong persistence guarantees.

Proceedings of the VLDB Endowment, Vol. 14, No. 9 ISSN 2150-8097.
doi:10.14778/3461535.3461543

To ensure data persistence, current KVSs write their data to devices with a block-based interface, i.e., SSDs or HDDs. However, the emergence of persistent memory (a.k.a., *PMem*, *NVRAM*, or *NVM*) promises byte-addressable data persistence with close-to-DRAM speed [14, 20, 50, 54]. Thus, leveraging PMem for KVSs and removing disk access has a large potential to improve KVS performance. It also supports the storage of arbitrary data structures without the need for record de-/serialization, which is required in traditional string-based KVSs.

To improve the performance of write-heavy workloads, most traditional persistent KVSs such as RocksDB [11] or LevelDB [13] optimize their inserts to avoid expensive write amplification on block-based devices. They employ log-structured trees [41] to collect records in-memory that are then written to disk in a single block-sized chunk. This approach requires additional disk-based write-ahead logging to ensure data persistence, as well as sophisticated merging logic for the disk-writes. Additionally, most disk-based KVSs require string or byte keys and values to store arbitrary data. This comes at a high de-/serialization cost for each access, significantly impacting the overall performance [12, 33].

Previous PMem research either focuses on how to adapt existing systems or develop new ones to harness PMem's potential. Various hybrid PMem-DRAM data structures have been proposed that leverage the speed of DRAM with the persistence of PMem for better overall performance. Most research focuses on the design of index structures, e.g., B-Trees [40, 55], LSM-Trees [29], or hash maps [30, 39]. Other research integrates PMem into larger systems, e.g., for database buffer management or recovery [2, 49]. Some simulated-PMem KVSs have also been proposed [29, 53].

However, as PMem has only recently become publicly available, the majority of previous PMem research uses simulations to estimate PMem performance in which key characteristics were assumed incorrectly [54]. These incorrect assumptions limit the effectiveness of proposed solutions as the optimal utilization of PMem requires knowledge of the underlying storage access patterns and characteristics. Recent research shows that Intel's Optane DIMMs [18] behave differently than DRAM and SSD [9, 54]. Thus, simply replacing disk-based storage with an identical PMem-based one does not yield the best performance. Benchmarks also show that sequential write latency to PMem is much closer to DRAM's performance, whereas there is a higher penalty for random reads than expected [14, 54]. This breaks one main assumption previous research built upon, that writes are slow and should be avoided and reads are fast and can be random.

To overcome the central performance issues of disk-based KVSs and incorrect assumptions of previous PMem research, we propose three PMem-specific access patterns for efficient data storage, *direct PMem writes, DIMM-aligned storage segments,* and *uniform thread-to-DIMM distribution*. We implement these patterns in *Viper*, a hybrid PMem-DRAM KVS whose persistence is built on PMem,

thus avoiding expensive disk accesses. Viper consists of a volatile index and persistent data, to perform most of the random operations in fast DRAM while optimizing the storage layout for efficient writes to PMem. In summary, we make the following contributions:

1) We propose PMem-specific access patterns to efficiently store and retrieve data directly to and from PMem in a hybrid PMem-DRAM environment.
2) We implement these access patterns in Viper, a hybrid PMem-DRAM KVS that persists its data directly in PMem.
3) We evaluate Viper against state-of-the-art KVSs and show that it outperforms them for core KVS operations. Viper exceeds existing PMem-only, hybrid, and disk-based KVSs by 4–18x for write workloads, while matching or surpassing their *get* performance.

The remainder of this paper is structured as follows. In Section 2 we cover some technical background that is relevant to our work. In Section 3 we introduce Viper and its core design principles. We show Viper's core functionality in Section 4, followed by a detailed evaluation in Section 5. We end this paper with an overview of related work in Section 6 and our conclusion in Section 7.

2 BACKGROUND

In this section, we introduce persistent memory and its performance characteristics, followed by a brief discussion of key-value stores.

2.1 Persistent Memory

Persistent memory (PMem) is an emerging class of memory devices that bridges the gap between DRAM and flash-based storage. It combines the byte-addressable data access offered by DRAM with the persistence of secondary storage while providing close-to-DRAM speed. Intel has recently made its *Optane DC Persistent Memory* DIMMs publicly available [18]. These DIMMs are based on 3D XPoint technology and are denser than DRAM DIMMs, offering larger capacities at a lower cost per GB ratio. Intel *Cascade Lake* CPUs support one PMem DIMM per memory channel, commonly resulting in six DIMMs per socket. As Optane is the only available PMem, we base our assumptions and designs in Viper on its characteristics and use PMem and Optane interchangeably.

For each DIMM, the integrated memory controller (iMC) maintains read and write pending queues (RPQs and WPQs) to buffer requests that were issued to the iMC. To guarantee persistence, the WPQs of the iMC are part of the asynchronous DRAM refresh (ADR) domain. Data that has reached a WPQ is therefore guaranteed to be flushed to PMem on power failure, whereas data in CPU caches is lost. While the iMC communicates with the DIMM in 64 Byte cache lines, physical media access occurs at 256 Byte granularity. Thus, an on-DIMM controller translates smaller requests to 256 Byte granularity before physically accessing the media, which causes read and write amplification. To reduce this effect, the controller combines adjacent writes using a write-combining buffer.

Running PMem in *App Direct Mode* gives the user explicit control over access to PMem, whereas *Memory Mode* uses PMem as a volatile extension of DRAM, in which DRAM acts as an "L4" cache and data is not persistent. To guarantee persistence in Viper, we use the App Direct Mode and map PMem into our application's virtual memory space via `mmap` [37] to leverage its byte-addressability.

Table 1: Bandwidth (in GB/s) and Latency (in ns) of DRAM, PMem, and SSD for 32 threads measured on our server[1].

	READ			WRITE		
	BW	Latency		BW	Latency	
	max	Seq	Rnd	max	Seq	Rnd
DRAM	100	40	190	70	110	170
PMem	40	50	450	13	230	900
SSD	1	115k	130k	1	125k	125k

PMem also supports interleaving data across DIMMs. While non-interleaved PMem appends the memory space of one DIMM behind the other, interleaving aims to improve the overall throughput of reads and writes by spreading sequential data accesses over multiple DIMMs in parallel, similar to RAID 0. This allows for better parallel access and a higher throughput [54]. In PMem, data is commonly interleaved in 4 KB chunks, distributing each consecutive block of 24 KB across all six available DIMMs. In Viper, we assume an interleaved PMem storage.

To guarantee persistence in App Direct Mode, the user must flush cache lines to PMem by using, e.g., the `clwb` (cache line write back) instruction. As the compiler and OS can re-order instructions for better performance, it is necessary to explicitly avoid this behavior by issuing an `sfence` instruction, which guarantees that the write to PMem was completed and not re-ordered [30, 50].

Even though PMem performance was assumed similar to DRAM but slower, recent research by Yang et al. [54] has found that the performance differs significantly and is less predictable. We perform high load micro benchmarks with 32 threads to get accurate latency and bandwidth numbers on our server for KVS workloads. As shown in Table 1, random reads have a 2.5x higher latency than DRAM, caused by a longer delay for accessing the media, but three orders of magnitude lower latency than SSD. At the same time, the latency of sequential persistent writes is almost equal to DRAM, since data only needs to reach the WPQ in the iMC to be persisted. Random writes suffer a 5x higher latency and are impacted heavily by the bandwidth. As a result, writing sequentially to PMem is encouraged, while random access should be directed to DRAM whenever possible. And while PMem's peak bandwidth is lower than DRAM's, it still reaches 40% for reads and 20% for writes, achieving significantly higher values than SSD.

2.2 Key-Value Stores

Key-value stores (KVSs) are a class of storage systems that handle data as ⟨ key, value ⟩ pairs. The basic operations KVSs implement are *put*, *get*, *delete*, and optionally *update* [5, 11, 24]. To access KVSs, two designs have emerged, *KVS servers* and *embedded KVSs*. A server-based KVS stores and synchronizes state that can be globally accessed by multiple applications running on different machines. It communicates with the applications via a network client/server API. Popular KVS servers are *Redis* [45] and *memchached* [35]. If the KVS is used by a single application, embedded KVSs provide a more lightweight alternative to server-based ones. These KVSs are embedded in the application and accessed using library function calls. Popular embedded KVS are RocksDB [11] and FASTER [5].

[1] 64 Byte access size; DRAM: Samsung M393A2K40CB2-CVF, PMem: Intel Optane DC 128GB, SDD: Micron 5100 ECO SSD

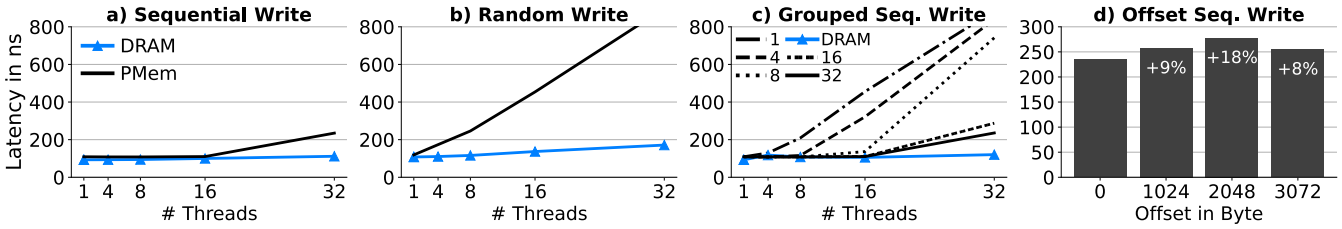


Figure 1: Write latency for various write patterns to DRAM and PMem.

A main advantage of KVS servers is that they are self-contained systems. This provides them with full system control, i.e., among others, they manage their own threads, concurrency, and I/O queues. However, this control entails an abstraction cost via, e.g., a network-based interface. On the other hand, embedded KVSs are controlled by the user within an application, which results in less communication overhead compared to network-based access and allows more fine-tuning. Yet, this control comes at the risk of incorrect usage, which might impact correctness and performance. To provide good performance and control, embedded KVS must design their interfaces as simple as possible without requiring the user to strictly follow patterns or complex procedures in case of, e.g., partial failures or system restarts. In this work, we focus on the design of such an embedded KVS and simple interface to allow the user to fully utilize PMem without high network overhead. We present such a design and implementation in Viper.

3 VIPER: A HYBRID KEY-VALUE STORE

In this section, we present *Viper*, a hybrid PMem-DRAM KVS that leverages PMem-specific access patterns for efficient data storage and retrieval. Viper avoids expensive disk access by persisting data in PMem while keeping an in-memory index to harness DRAM's lower random access latency over a fully PMem-based approach. We first discuss our hybrid design in Viper in Section 3.1 followed by a description of Viper's core components in Section 3.2.

3.1 Hybrid Design

To fully utilize both DRAM's and PMem's strengths, we propose a hybrid storage approach in Viper. Viper consists of a volatile hash index located in DRAM and persistent data blocks located in PMem. While Optane DIMMs can act as a drop-in replacement for SSDs to achieve data persistence, to fully leverage the performance of PMem, we need to understand its storage layout and beneficial access patterns. All data is durably stored in persistent memory and the hash index contains only references to the storage location.

Hybrid storage models have also been proposed in previous work on index structures [40, 55] under the concept of *selective persistence*. The idea behind selective persistence is to store only the data required to rebuild the entire system state in persistent memory and keep a dynamic recoverable state in volatile memory. Viper is designed to be an embedded KVS similar to RocksDB [11] or FASTER [5] and not a KVS server. Thus, users interact directly with the database in the same process without any network interface.

PMem Access Patterns. Initial studies on real PMem show complex performance characteristics, which often lead to low bandwidth and high latency [20, 54] In Viper, we propose three core design choices for PMem-specific access patterns that significantly impact its performance on real hardware:

1) **Direct PMem writes.** As sequential PMem writes are faster than previously assumed in simulations, Viper writes all data directly to PMem without an intermediate DRAM buffer.
2) **Uniform thread-to-DIMM distribution.** Viper minimizes the thread-to-DIMM ratio for inserts by assigning threads to different memory regions.
3) **DIMM-aligned storage segments.** Viper stores data in DIMM-boundary aligned *VPages* to balance DIMM contention with parallelism. Smaller VPages result in more threads accessing the same DIMM and larger VPages result in a single thread accessing multiple DIMMs, both leading to a worse, and thus disadvantageous, thread-to-DIMM ratio [54].

We demonstrate the impact of these design choices in Figure 1 (see Section 5.1 for our system setup). We perform 64 Byte stores followed by `clwb` and `sfence` with a varying number of threads in PMem and DRAM. Figures 1a and b show that sequential writes have a similar latency for PMem and DRAM (maximum 2x higher for 32 threads), while random writes perform significantly worse on PMem even for low thread counts. This is due to Optane's internal write-combining buffer, which combines adjacent writes to reduce expensive media flushes but cannot combine small random writes, causing high write amplification. From this observation, we derive our *direct PMem writes* design.

Figure 1c shows the importance of an even distribution of threads across all DIMMs. We distribute the threads across k memory regions (1 GB each), representing log files, to which they write sequentially. Using 1 log file (denoted as 1 in the plot), all threads write adjacent cache lines, i.e., thread 1 writes bytes 0–63, thread 2 writes 64–127, and so on. When using the same number of threads and logs, each thread has its own disjoint memory region. With more logs, fewer threads share a memory region and evenly distribute across the DIMMs. The poor performance of 1 log is caused by all threads operating on a single DIMM (32×64 Byte = 2048 Byte) and thus, disregarding the inherent parallelism of interleaved PMem. We see a performance increase when using more logs as the threads profit from PMem's parallelism by writing to varying locations evenly distributed across DIMMs. From this observation, we derive our *uniform thread-to-DIMM distribution* design.

Finally, Figure 1d shows the impact of storage-aligned access. In this benchmark, we let each thread write 4 KB sequentially and alter the alignment of the writes. We see that 4 KB aligned writes (offset = 0) achieve the lowest latency, while a 2 KB offset has an 18% higher latency. This is again caused by the necessity of accessing two DIMMs to write 4 KB instead of only one. From this observation, we derive our *DIMM-aligned storage segments* design.

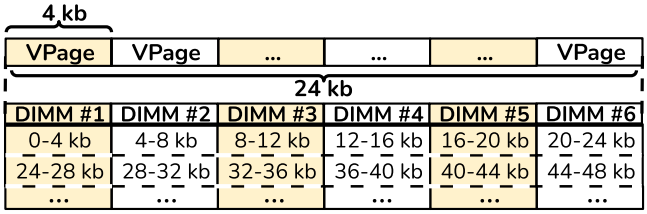


Figure 2: Viper's storage aligned with 4 KB PMem layout.

Volatile Index. Our evaluation of real PMem hardware shows that random operations have a significantly higher latency than sequential ones and achieve lower bandwidth (cf. Figure 1 and Table 1). Thus, we avoid (possibly multiple) expensive random operations to the hash index by locating it in DRAM. Additionally, the efficient design and implementation of hash maps in DRAM are widely studied [22, 25, 34, 48], allowing us to fully take advantage of these concepts. Persistent hash maps, on the other hand, have only recently been introduced [30, 38, 39, 46] and show lower performance than DRAM-based ones. Furthermore, due to the persistence of every operation in the map, complex logic is required to avoid concurrency and memory issues, e.g., persistent memory leaks, invalid pointers, and blocked persistent locks. For our implementation of Viper, we build on *CCEH* [38] and use it in DRAM instead of PMem. CCEH uses an extendible hashing approach, thus allowing for dynamic resizing without an expensive full table rehashing. As we use the volatile index to store offsets to PMem locations, we refer to it as *Offset Map* in the remainder of this work.

Persistent Data. As our goal is to persist all data in Viper, we need to store all key-value pairs on a durable storage medium. We choose Intel's *Optane DC Persistent Memory* [18] in our implementation. In Viper, we write all records directly to PMem-based storage segments (design choice 1). Viper's main storage segments are called *VPages* and contain the individual key-value records as well as some metadata. Figure 2 shows how we align VPages with the layout of the underlying PMem DIMMs (design choice 3). As described in Section 2.1, we assume a system configuration with six DIMMs per socket. However, Viper is configurable to work on any number of DIMMs. We use Optane DIMMs in the interleaved mode to achieve a higher degree of parallelism [54]. In the interleaved mode, data is striped across all DIMMs in 4 KB pages. We exploit this striping by aligning VPages to the 4 KB page boundaries. This allows us to access exactly one DIMM per VPage, thus reducing contention on the DIMMs during parallel access (design choice 2) .

3.2 Architecture

Viper consists of three main components, persistent *VBlocks* and *VPages*, as well as an in-memory *Offset Map*. We show Viper's core components in Figure 3. On the right-hand side, we see Viper's persistent storage segments (VPage) grouped into VBlocks, located in PMem. On the left-hand side, we see Viper's volatile Offset Map, which acts as an index by storing the key and persistent storage location of each record. In the remainder of this section, we describe the design of the three core components in detail for fixed- and variable-sized records. We first describe the VBlock and Offset Map, as these are identical for both variations, followed by the fixed-sized VPage design and the variable-sized modifications.

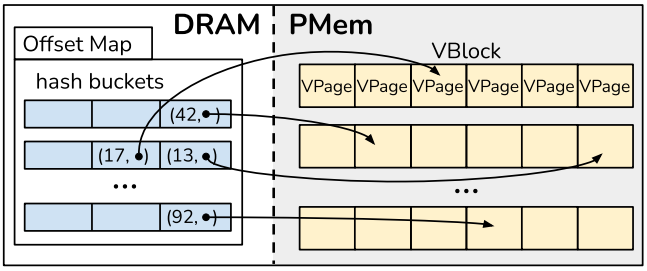


Figure 3: Viper's architecture. VPages store key-value records in PMem (right). The Offset Map stores (key, record-offset) entries in a volatile hash index (left).

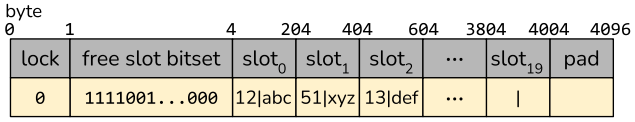
3.2.1 Common Components. In this section, we present components that are identical for fixed- and variable-sized records: the *VBlock* and *Offset Map*, as well as Viper's metadata management.

VBlock. In Viper, we align VBlocks to the boundaries of the underlying interleaved set of DIMMs, spanning exactly 24 KB. Each VBlock contains a fixed number of VPages, one VPage for each DIMM, stored in an in-place array for efficient access. VBlocks contain no logic themselves but simply act as a grouping of VPages to reduce the bookkeeping overhead in Viper. Each VPage is 4 KB (DIMM-aligned) and contains some metadata plus the actual key-value records stored in *slots*. They are the actual storage units in Viper. To support larger key-value pairs, Viper scales VPages to multiples of 4 KB and VBlocks to multiples of 24 KB, ensuring the same 4 and 24 KB alignment. For simplicity, we assume 4 KB VPages and 24 KB VBlocks in the remainder of this work.

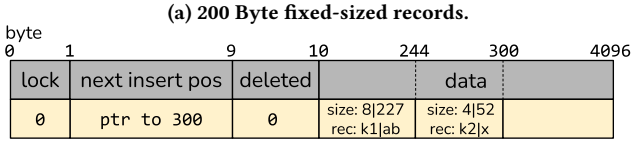
Offset Map. The *Offset Map* is the core volatile index that Viper uses to keep track of all records. In Viper, the Offset Map is an in-memory, concurrent hash map. When a record is inserted into Viper, it is first persisted in a VPage and then the *offset* of the record is stored as the value in the Offset Map for the given key. The *offset* consists of three parts: the VBlock id, the VPage id, and the record position in the VPage. The record position depends on fixed- or variable-sized records. With these three parts, Viper can uniquely locate any given record. Viper stores the offset in a 64-bit `Offset` object, where the most significant 45 bits represent the block id, the following 3 bit represent the page id, and the next 16 bit are used for the record position. The bit-assignments may be modified in case the user has specific knowledge of the expected workload, e.g., very large records or the number of DIMMs varies significantly.

Analogously to previous work, we use *fingerprinting* in order to store keys larger than 8 Byte in the Offset Map [30, 40]. Instead of storing the actual key in the map, Viper stores the hash of that key and checks for equality only if the hash matches. This significantly reduces the number of expensive comparisons with the keys in PMem, as very few collisions are expected for 64-bit hashes.

Metadata Management. To grow, Viper allocates VBlocks in PMem and maps them into the virtual memory space via `mmap` [37]. To keep track of the virtual addresses, Viper stores a pointer to each VBlock in a list in DRAM. This allows for easy access to an arbitrary VBlock by its implicit id, which is equal to the offset in the list. Once the available VBlocks reach a certain configurable filling degree, Viper allocates additional VBlocks and adds them to the list. Viper supports PMem allocation from both *devdax* or an *fsdax* directory. Data is allocated in increasing memory order



(a) 200 Byte fixed-sized records.



(b) Variable-sized records.

Figure 4: VPage layout with example entries. Key-value records are stored consecutively.

(devdax) or increasing file names (fsdax) to guarantee ordering, thus maintaining the VBlock order after a restart. To reduce the number of memory allocations, large chunks (or files) are allocated, which contain 43690 VBlocks by default (1 GB). Metadata recovery and mapping all data back into Viper's virtual memory space takes only a few milliseconds, as it mainly consists of `mmap` calls.

3.2.2 Fixed-Sized Records. We now present the VPage design for fixed-sized records, as shown in Figure 4a.

VPage Data. Viper stores the actual key-value records in VPages. Both the key and the value are stored together in a single *slot*. The slot id is used as the third part of the Offset Map entry (*record position*) for fixed-sized records. When using the term key-value *record*, we refer to both the key and value together. The number of slots per VPage depends on the record size, where larger records require more space and thus fewer fit into the available 4 KB. Viper uses nearly all of the 4 KB to store data, as only a few bytes are needed for metadata. We describe the calculation for the number of slots with the metadata size below.

VPage Metadata. The metadata is stored in the first few bytes of the VPage. It consists of a version lock byte and a bitset indicating which slots are free or populated. Both concepts are also used in previous research on PMem data structures, e.g., in tree nodes [6, 40] or in hash buckets [30]. We use a lock byte to handle concurrent access to the VPage, allowing only one thread to concurrently modify its data. The lock is acquired and released via atomic compare-and-swap operations (CAS). We thus avoid the use of heavy-weight mutexes at this point. Even though there are persistent CAS implementations [52] that ensure correct persistence-semantics, Viper uses regular in-memory CAS operations with less overhead. The lock is only relevant during active use and is reset after a crash.

The bitset contains k bits, one for each slot in the VPage. A set bit indicates that the slot is occupied and contains data. An unset bit, in reverse, indicates that the slot is free. This allows Viper to efficiently delete a record by setting the bit at its slot position to 0.

VPage Slot Count and Metadata Size. The exact size of the metadata depends on the record size, as Viper requires one bit per slot in the bitset. To determine the metadata size, we first calculate the number of slots per page by dividing the VPage size by the record size, $\lfloor size_p / size_r \rfloor = num_{slots}$. This is rounded down to the nearest integer as we cannot have partial slots. To avoid an over-allocation of the VPage, we need to check if the metadata still fits. The metadata size is calculated as $1 + \lceil num_{slots}/8 \rceil = size_m$ bytes, for the lock + bitset. We round up the bitset size, as the underlying system cannot work on individual bits but requires full bytes. If the data plus metadata is too large for the VPage ($num_{slots} * size_r + size_m > size_p$), we reduce the number of slots by one. All unused space at the end of the VPage is left as padding to keep the 4 KB alignment.

3.2.3 Variable-Sized Records. To support variable-sized records, the VPage-design needs to be modified, as shown in Figure 4b.

VPage Data. Records are not stored in fixed slots, as their size is unknown a priori. Thus, Viper uses all non-metadata bytes in the VPage as a log. Each record is consecutively written to this log together with the respective key and value length. The sizes are stored in a single 32-bit value (15 bits for key, 16 bits for value) to allow for atomic updates. The least significant bit of the value indicates whether the record is set (= 1) or deleted (= 0). The offset in the log is used as the third part of the Offset Map entry (*record position*) for variable-sized records. For key-value pairs larger than 4 KB, Viper dynamically uses an entire VBlock as a single VPage. For even larger records, Viper writes the record across multiple large VPages and marks these as *overflow* pages. Thus, large records do not impact the design of Viper, as it still has unique VBlocks per client and equal distribution of client threads to DIMMs.

VPage Metadata. As the VPage does not contain any slots, the free slot bitset is removed. Instead, each VPage now contains a pointer to its next insert position, i.e., the tail of the log, and an 8-bit integer to track how much data has approximately been deleted (i.e., metadata bit = 0) and needs to be compacted. The metadata size is fixed for variable-length records at 10 Byte, allowing for 4086 Bytes of records per VPage.

4 KEY-VALUE STORE OPERATIONS

In this section, we discuss the common KVS operations *put* (Sec. 4.2), *get* (Sec. 4.3), *update* (Sec. 4.4), and *delete* (Sec. 4.5), as well as space reclamation (Sec. 4.6) and the recovery of an existing database (Sec. 4.7). Before discussing the operations, we present the *Viper client* through which users interact with Viper.

4.1 Viper Client

Commonly in embedded KVSs, a database handle is created for a given file, which either creates a new database if the file does not exist or opens the existing database. This handle can be used by multiple threads to interact with the KVS, by issuing, e.g., *get* or *put* requests. However, when the KVS does not control or own the threads, the handle has to control external concurrent access. Examples for such embedded KVSs are RocksDB [11] or LevelDB [13]. For *put* requests, this means providing a new insert location for each request. This central synchronization point quickly becomes a bottleneck, which we avoid in Viper by introducing a *Viper client*.

As Viper is an embedded KVS, the client does not contain any network logic as common in KVS servers. It is a light-weight object that exposes the KVS-operation interface to the user and contains information on where to write future records to reduce synchronization within Viper. In Figure 5 we show clients interacting with Viper. In our example, three clients have been created. Each client is initialized with its own VBlock (#1 $\rightarrow$ VBlock_0, #2 $\rightarrow$ VBlock_1, #3 $\rightarrow$ VBlock_2), i.e., no two clients write new records to the same VBlock. To indicate that a VBlock is currently "owned"

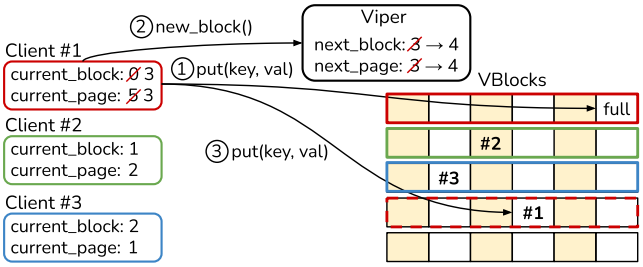


Figure 5: Client requests new VBlock. Client #1 requests a new VBlock after a *put* to a full one and then writes to the new VBlock.

by a client, an owned_bit is set in the version lock of the first VPage. This bit is used for space reclamation and recovery. The client then writes data to its current VBlock/VPage and progresses the VPage until all VPages are full. Once a client cannot *put* data into its VBlock because it is full ①, it requests a new VBlock from Viper ②. Viper then returns the next block to the client and atomically updates the next_block and the next_page counters. Viper stores the next_block and next_page counters in a single 64-bit variable that can be updated atomically with a compare-and-swap operation. The next_page counter is chosen randomly to achieve a uniform distribution across DIMMs. When the client receives its next block, it updates its references and inserts the record into the new VPage ③. This approach significantly reduces the coordination overhead within Viper, as it does not need to issue a new write location for each *put*. If a VBlock fits, e.g., 100 records, the overhead is reduced by 100x, as a client only needs a new location every 100 writes.

As Viper supports space reclamation (Sec. 4.6), it also keeps track of free blocks in a concurrent queue. If a free block is present in the queue, it is given to a client for re-use rather than allocating a new client. In that case, the block is removed from the queue and the next_block counter remains unchanged.

4.2 Put

To insert data into Viper, clients must issue a `put(Key, Value)` request. The pseudo-code for this is presented in Listing 1. The client first acquires the VPage lock for its current VPage in a blocking call (Line 1). To acquire exclusive access, the version lock is atomically compare-and-swapped with a +1 increment to an odd-number, e.g., from 0 to 1. If a client encounters an odd-numbered lock, it retries its operation. Once the client has exclusive access to the VPage, it searches for the next free slot (Line 2). If the VPage is full, the client releases the lock, updates its page and block information, and retries the *put* operation (Lines 3-6). To update the page and block information, it either progresses to the next VPage in its current VBlock or it requests a new VBlock from Viper.

If there is a free slot in the current VPage, the client stores the record in the free slot and persists it (Lines 8-9). To write the current cache line to PMem, the `Persist` method issues a `clwb` call to the underlying system followed by an `sfence` call. The `sfence` enforces correct ordering guarantees, i.e., after the call, the data is guaranteed to be persisted. Only after the data is persisted does the client update and persist the bitset (Lines 10-11). The order here is important, as the bitset becomes the ground truth for recovery [30, 40]. If the bitset indicates a populated slot but the data is not properly stored, Viper is in an inconsistent state.

Listing 1: Viper's `put(Key k, Value v)`

```
1  AcquireVPageLock(v_page);
2  free_slot_idx = FindFreeSlot(v_page.slot_bitset);
3  if (free_slot_idx == max_bitset_size) {
4    ReleaseLock(v_page); GetNewVPageOrVBlock();
5    return Put(k, v);
6  }
7
8  v_page.slots[free_slot_idx] = {k, v};
9  Persist(v_page.slots[free_slot_idx]);
10 v_page.slot_bitset[free_slot_idx] = 1;
11 Persist(v_page.slot_bitset);
12
13 offset = {block_num, page_num, free_slot_idx};
14 [is_new, old_offset] = offset_map.Insert(k, offset);
15 if (!is_new) DeleteOldRecord(old_offset);
16
17 ReleaseLock(v_page);
18 return is_new;
```

Once the data is persisted, the client inserts the new offset into the Offset Map. If the Offset Map contained an entry for the key, the old value is overwritten and the client must ensure that the record at the old location is deleted by setting the corresponding bit to 0 (Line 15, see Section 4.5). As the Offset Map handles concurrency, it guarantees that in the event of concurrent writes to the same key, one client will see the value added by the other client as an old offset, thus deleting the other client's value. Finally, the client releases the lock on the VPage and returns a Boolean indicating whether a new key was inserted or an existing one was overwritten (Lines 17-18). The lock is released by atomically storing another +1 increment, thus, making the lock even-numbered again.

Crash Consistency If a crash occurs between persisting the bitset and the deletion of an old record, Viper contains two values for the same key. To guarantee a deterministic recovery and thus ensure atomic writes, Viper selects the greater ⟨ block_id, page_id, slot_id ⟩ in case of a conflict. We note that this is not necessarily the newer value, as "old" block ids are reused after reclamation but it constitutes a deterministic tie-break during recovery. To ensure that the new value is not read until it is guaranteed to be deterministically recoverable, clients hold the VPage lock until the old record is deleted. In rare cases, this may lead to a deadlock, as two clients might need to lock the same two VPages in reverse order. If a deadlock is detected, i.e., the lock cannot be acquired in x tries, the client adds the offset it needs to delete O to a global list. All clients in the deadlock continuously check this list for offsets O' that match their current VPage, delete the record at O', and remove it from the list. If a client notices that O was deleted from the list, the deadlock is solved and it can return after unlocking its VPage. In a micro benchmark with 50 million mixed operations, Viper encounter only two such deadlock-like scenarios.

Variable-Sized Records. Inserting variable-sized records follows a similar procedure as shown in Algorithm 1, but the actual writing of the data is different. To insert a variable-sized record, the client first retrieves the `next_insert_position` from the VPage metadata. It then writes the record to PMem at the given location followed by a `Persist` call. Only then does it write the record's metadata in front of the record. This order guarantees that if the metadata is present, the record is persistently stored. This is identical to persisting the bitset after the slot for fixed-sized records. When a record does not fit into a page, the client checks if the

key without the value fits. If it fits, the value is written to the next page, and only then is the key written with metadata indicating a value length of 0, which tells Viper that the value is stored on the following page to ensure the same persistency guarantees as above. If the key does not fit, the record is written to the next page and the metadata is set to an invalid configuration on the current page, indicating that no more data is present after this marker. After inserting the record, the `next_insert_position` is updated to reflect either the end of the page or a new position.

4.3 Get

To retrieve individual records from Viper, the client issues a `get(Key)` request. To efficiently scale for read-heavy workloads, Viper uses lock-free reads [6, 30]. First, the client searches for the key in the Offset Map and returns an error if not entry was found. The client then atomically reads the version lock of the VPage that contains the record into $l1$. If $l1$ is odd-numbered, another client currently holds an exclusive lock and the entire read is retried, as a VPage modification might have altered the retrieved offset. In an unlocked state, the client reads the value at the given offset. The pointer retrieval is a lookup in Viper's VBlock list for the offset's block id, followed by direct accesses into that VBlock's page list at the page id and the VPage's slots at the slot id. We note here that the VPage array within a VBlock and the slots within a VPage are known at compile-time, thus allowing the compiler to combine the latter two lookups into simple pointer arithmetic on the VBlock pointer from the initial lookup. Before returning the value, it again atomically loads the version lock into $l2$. If $l1 = l2$, the VPage was not modified and the value can be safely returned. If $l1 \neq l2$, the entire read operation is retried, as a conflict might have occurred.

Retrieving variable-sized records follows the same steps, but the actual record lookup differs slightly. Instead of reading a record from a given slot, it first reads the record length at the given offset in the VPage log and then retrieves the value according to its size.

4.4 Update

In order to update a value in Viper, the user can call `update(Key, UpdateFn)`, where *UpdateFn* is an arbitrary function that receives a value and modifies it atomically. As Viper does not copy the values, modifications are made in-place in PMem. To avoid partial update anomalies, only atomic updates can be performed. However, this allows the user to modify up to 8 Byte (or 16/32/64 with modern AVX-512 CPUs) of a value in-place. This is useful to, e.g., update counters or other individual fields in the value [3]. Updating in Viper is similar to *get* but instead of returning the value if no version conflict occurred, the client acquires an exclusive lock and applies the *UpdateFn* to the value. Thus, any subsequent operations are aware that a modification was performed.

For non-atomic updates, the value must be re-inserted. To achieve this, the user *gets* the value, creates a copy, modifies it, and finally calls *insert* for the same key with the new value. This is a common approach in many KVSs [11, 13, 21, 31] and Viper always falls back to this approach if in-place modifications are not possible. This is also the approach for variable-sized records, as modifications in them might change their size. In Viper, records are tightly packed in the log and do not allow for any subsequent size variation.

Two main advantages of in-place updates over conventional copy-on-write are avoiding serialization and fewer cache line flushes, i.e., only one `Persist` call is needed in Viper as no metadata is updated. Also, recent work shows that in-place modification is preferred over copy-on-write for PMem [23, 51].

4.5 Delete

To delete a record, the client issues a `delete(Key)` request. The client first looks for the key in the Offset Map and returns false if it is not found. If it was found, the client retrieves the VPage from the offset information and acquires its lock to block other modifying access. In Viper, the actual record is not erased, but the corresponding free slot bit is set to 0 and the bitset is persisted to make the deletion durable. Then, the key is removed from the Offset Map before releasing the page lock and returning a successful deletion. For variable-sized records, the deletion bit is not set in the bitset but rather in the record metadata in the log. The size information is not modified, as it is required to skip the deleted record when scanning the VPage during recovery or compaction.

4.6 Space Reclamation

After various records have been deleted or re-inserted, the VPages contain many free slots or tombstoned records in the log. In order to reuse this free space, Viper runs a periodic background space reclamation process. In this reclamation, Viper scans the bitsets of the VPages to see how many free slots are available. If the number of free slots in a VBlock is higher than a configurable threshold and the VBlock is not currently "owned" by a client, the VBlock is compacted into a new VBlock, marked as free, and added to the free block queue. Compacting a VBlock is equivalent to re-inserting each record in that VBlock. Thus, when compacting many VBlocks, the records are tightly packed again. If a client reads a record that is currently being compacted, it either reads the stale offset and retries because the version lock of the compacted VPage has changed or it reads the new offset. Each VPage is locked for the entire duration of its compaction to avoid modifications throughout.

For variable-sized records, Viper checks the metadata of each VPage for the approximate free space on this page. If the VBlock reaches a configurable threshold, it gets compacted as in the fixed-sized process. After the compaction of a VBlock, it is marked as free with a *free* bit in its first VPage's lock byte. This allows Viper to recognize free VBlocks during a recovery. This process can also be used to deallocate VBlocks at the tail of the VBlock list and thus reduce its PMem footprint after many records have been deleted.

4.7 Recovery

A persistent KVS needs to be able to recover from a crash or be re-opened after a regular shutdown. In Viper, we handle both scenarios identically, as all required metadata is continuously persisted during its normal operational mode. Viper stores a small amount of metadata in PMem to keep track of the number of allocated VBlocks, the number of used VBlocks, and the total memory-mapped size. Every time new VBlocks are allocated in PMem, the metadata is updated to reflect the total number of allocated blocks. Additionally, every time a new VBlock is assigned to a client, the number of used blocks is incremented in the metadata.

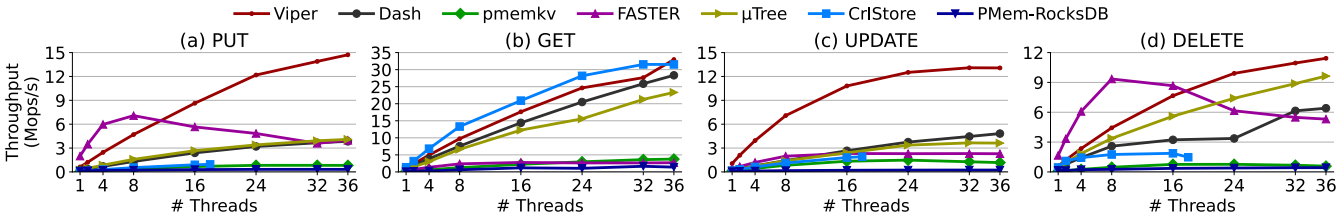


Figure 6: Core KVS operations.

When Viper is opened with an existing database, it checks this metadata and prepares for a recovery based on it. Viper maps existing VBlocks into its virtual address space and stores pointers to each VBlock, as described in Section 3.2. After mapping all VBlocks, Viper checks for the number of used blocks and scans those to retrieve the records in them. For each VPage, Viper checks which slots are set (fixed-sized) or scans the log for non-deleted records (variable-sized) and inserts the offsets into the map. This can be parallelized by assigning disjoint VBlock-ranges to different threads. After scanning all VBlocks, the next_block counter in Viper is updated to the highest used block_id + 1, so that new clients receive fresh VBlocks (see Section 4.1).

5 EVALUATION

In this section, we present the evaluation results of our implementation of Viper compared against other KVSs. In Section 5.1 we describe our setup, followed by an introduction of the other systems in Section 5.2. We present our Micro-Benchmark results in Section 5.3 and our YCSB evaluation in Section 5.4.

5.1 Setup and Methodology

We run all experiments on an Intel Xeon Gold 5220S CPU server and pin all threads to one socket to avoid cross-socket data access. The CPU has 18 cores (36 logical cores via hyperthreading). The socket is connected to 750 GB PMem, in six 128 GB Intel Optane Persistent Memory DIMMs, and to 96 GB DRAM. To access the Optane DIMMs directly, we use *devdax* mode. We prefill the stores with 100 million records before performing the benchmark operations and use 16 Byte keys (e.g., a UUID) and 200 Byte values, as these represent common sizes in real-world KVSs [3].

We implement our prototype of Viper in C++, compiled with GCC 9.3 on Ubuntu 20.04. We use and modify the CCEH map [38] for the offset map and low-level libpmem (v1.10) [42] calls to persist data in PMem. Our code is open-source and available on Github[2].

5.2 Other Systems

We evaluate Viper against six other systems to show the impact of various design choices in Viper: *FASTER*, *pmem-rocksdb*, *Dash*, *pmemkv*, *µTree*, and *Cross-Referencing Logs*. *FASTER* [5] (v1.8.0) is a state-of-the-art embedded hash-based KVS, which we run backed by PMem instead of SSD, making it a hybrid DRAM-PMem system. We initialize FASTER's hash index identically to the authors' evaluation with $\sim$ $\#keys/2$ hash buckets, resulting in a 2 GB index. We set the log size to 6 GB, which is ~1/4 of the total raw data size. *pmem-rocksdb* [32] is a modified version of RocksDB to work explicitly with PMem by optimizing SSTables for and placing the WAL on it. We run pmem-rocksdb with the same configuration as the authors. These comparisons show the need for new PMem-aware designs instead of drop-in replacements and minor modifications.

We also compare Viper against two PMem-only setups to show the benefit of a hybrid design. As proposed in previous work, index structures can be used together with a persistent allocator as a KVS [30, 40, 55]. *Dash* [30] is a state-of-the-art PMem-optimized hash index that we pair with PMDK's persistent allocator [42]. A second PMem-only system we evaluate is Intel's hash-based *pmemkv* [43] (v1.4), which we run with the *cmap* backend [19, 44].

We also evaluate Viper against two hybrid PMem-DRAM systems. *µTree* [6] is a state-of-the-art hybrid BTree implementation that natively supports large values, making it suitable for a KVS use-case. We note that the performance of a BTree is expected to be slightly lower for single record operations, due to sorting overhead for additional range-query support. *Cross-Referencing Logs* (CRL) [17] were proposed to bridge the gap between volatile and persistent KVSs by persisting cross-referencing logs between two KVSs, one in DRAM and one in PMem. As CRL is not publicly available, we implement it (*CrlStore*) with Intel's volatile TBB concurrent hash map as the DRAM KVS [19] and the persistent map as the PMem KVS [44], which both fulfill the per-record locking requirements of CRL. As CRLs require front- and backend threads, we use a 1:1 mapping for all write operations, limiting our results to 18 threads in the plots. We do not employ a dynamic mapping, as proposed by the authors, because the backend threads constitute the bottleneck in our experiments. For *get* requests, we use only frontend threads.

5.3 Micro Benchmarks

In this section, we evaluate Viper's performance through various micro benchmarks. To this end, we discuss the performance of the four core KVS operations, the impact of different record sizes and variable-length records, followed by the systems' memory consumption. We then evaluate Viper-internal design choices by showing the impact of in-place updates and of data placement on DRAM or PMem, followed by an operation breakdown, space reclamation impact, and recovery performance.

5.3.1 Key-Value Store Operations. To understand the throughput of Viper, we compare it against the other KVSs for the core KVS operations *insert*, *get*, *update*, and *delete*. We initially fill each KVS with 100 million 216 Byte records (16 B key, 200 B value), before performing 50 million individual operations on them. Each client inserts consecutive keys from a disjoint range. For update, get, and delete, we uniformly choose a random key in each call. We use a

[2]https://github.com/hpides/viper

fresh KVS for each operation to avoid unintentional caching effects. For all *get* operations, we explicitly read the value to ensure that it is accessed from the underlying medium and not just pointed to.

The results are shown in Figure 6. In (a), we see that Viper's insert throughput scales well with the number of threads due to its efficient sequential access across multiple DIMMs via the Viper clients, reaching a peak of 15 million puts/s with 36 threads. The PMem-optimized Dash and µTree also scale but achieve only ~4 Mops/s. Both are limited by the record allocation outside of the actual index structure, which shows the need for a more structured insert mechanism. FASTER performs better than Viper for few threads, as the data is initially written to DRAM and is not persisted. However, after 8 threads its performance decreases. Once FASTER's DRAM-based log is full, it writes old segments to PMem to free space. This becomes a bottleneck, as the log needs to wait until the segment was copied and flushed before it can allocate a new segment to write to. The other systems do not scale well and achieve fewer than 1 million inserts/s due to unoptimized random hash map operations performed in PMem.

Retrieving records (b) is split into two groups. FASTER, pmemkv, and RocksDB do not perform well for random *get* request due to inefficient lookups in PMem, all peaking below 4 Mops/s. FASTER and RocksDB are optimized for access from disk-storage, disregarding random access capabilities PMem, while pmemkv is built for PMem but with an unoptimized hash index. The other group of systems are optimized for PMem and achieve peaks between 25 and 35 Mops/s. This shows that *get* performance heavily depends on the chosen (hash-) index implementation and that the DRAM-based index in Viper does not significantly outperform PMem-based Dash. We plan to investigate the use of different index types in Viper in future work, as a recent study shows that, e.g., Dash achieves significantly higher lookup rates than CCEH [16]. In Figure 11, we show that DRAM-based Viper achieves ~50 Mops/s, indicating that CrlStore is limited by the TBB concurrent map in this evaluation.

In real world use-cases, record updates are often small modifications, e.g., 8 B counter updates [3]. In such a workload (c), we see that Viper outperforms all other systems due to its atomic PMem-aware in-place modification compared to the read-modify-write semantics of the other systems. We discuss the different update semantics in Viper in more detail in Section 5.3.5.

Deleting (d) records behaves similarly to updating in Viper, as it performs a key lookup followed by a small write, i.e., invalidating the slot. Other systems' delete performance is higher than their respective update performance, as many use a tombstone invalidation without the need to insert a new entry.

Our evaluation shows that for inserts, a PMem-specific sequential write pattern considerably improves the performance over batched disk-based approaches or random PMem allocations by 4–18x. Also, the update performance of Viper is superior, as it can perform in-place updates in persistent storage, which other systems cannot. For data retrieval, Viper performs on par with comparable systems. As PMem-RocksDB performs worse than all other systems, we omit it from future evaluation due to limited space.

5.3.2 Key-Value Record Size. To understand the impact of record sizes, we evaluate all systems with varying key and value lengths. We evaluate the impact of very small records (8 Byte key, 8 Byte value), more common sizes (16 B, 100 B) and (32 B, 500 B), as well as large records (100 B, 900 B). We define a fixed prefill data size of 20 GB, which we divide by the record sizes to get the number of records to prefill each system with, i.e., 20 $GB/16\ B = 1.25$ billion 16 B records and 92/37/20 million 216/532/1000 B records. We then insert 10 GB in the same manner, i.e., exactly half as many records as the prefill. In a second workload, we issue 50 million *get* requests on a prefilled KVS. All runs are performed with 36 threads. We omit µTree, as it does not support large keys.

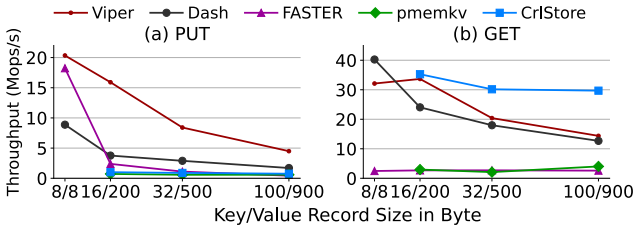


Figure 7: Key-Value size impact.

The results are shown in Figure 7. For 16 Byte records, Viper achieves ~20 M puts/s and decreases linearly with an increasing record size, as it becomes PMem bandwidth-bound. FASTER also achieves nearly 20 M puts/s for 16 B records, as many of them fit into the DRAM-based log and are not persisted. With increasing record sizes, FASTER's performance drops to under 2 Mops/s as fewer records fit into the log, requiring more frequent PMem flushes. From this result, we see that efficient access patterns to PMem, as employed in Viper, have a higher impact on the overall performance than simply reducing the number of PMem flushes, as done in FASTER, via a DRAM buffer followed by a large PMem flush. We note that especially for larger records, the impact of a single additional metadata flush decreases, as multiple flushes are required for the record alone.

Dash benefits from 16 Byte records, as it does not require an extra memory allocation outside of the index. However, its insert performance is only about 50% of Viper's, which demonstrates the high overhead of random writes to PMem over sequential ones. For larger records, random memory allocations become the bottleneck in Dash. Both pmemkv and CrlStore cannot insert the 1.25 billion 16 B records as they run out of memory. We note that this behavior is expected, as explained in the PMDK documentation: "allocations of a size less than 64 Bytes [are] extremely inefficient and discouraged."[3] Thus, both pmemkv and a default allocator KVS are not suitable for small records, and for larger records, they are limited by their inefficient PMem writes.

The *get* performance trend of Viper is similar to the insert performance, where access to larger records is bandwidth-bound. Surprisingly, 16 B *gets* are less efficient than 216 B, as CCEH performs better with fewer entries. Dash retrieves 16 B records very efficiently, as the values are stored directly in the map without indirection. For larger records, its performance is also bandwidth-bound. CrlStore exhibits a consistently high *get* performance, as all requests are answered from DRAM without PMem access. Both FASTER and pmemkv show the same low performance as in the previous section due to inefficient access.

[3] https://pmem.io/pmdk/manpages/linux/v1.8/libpmemobj/pmemobj_alloc.3

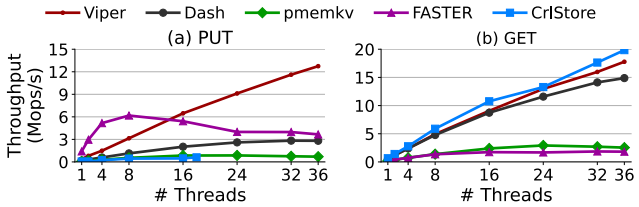


Figure 8: Variable-sized ~216 Byte records.

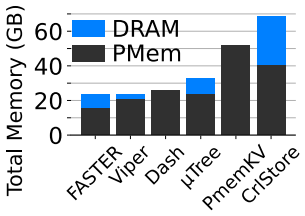


Figure 9: Total memory.

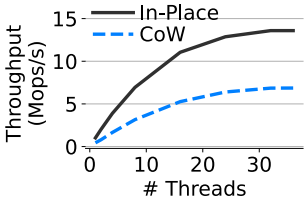


Figure 10: Update strategy.

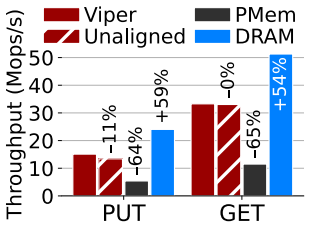


Figure 11: Viper versions.

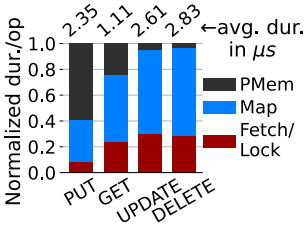


Figure 12: Op breakdown.

5.3.3 Variable-sized Records. In this benchmark, we evaluate the impact of variable-sized records on the performance of the systems. To this end, we prefill 100 million records of about 216 Byte, with a normal distribution around 16 B for the key ($\mathcal{N}(16, 3.2^2)$) and 200 B for value size ($\mathcal{N}(200, 40^2)$). We then perform each 50 million puts and gets and measure the throughput.

The results shown in Figure 8 are in line with those of the core operations (cf. Fig 6). For *puts*, Viper clearly outperforms the other systems due to its efficient VPage design. Record retrieval also follows the same trend of fixed-sized records discussed above. However, both *put* and *get* achieve lower overall throughput compared to fixed-sized records. For fixed-sized records, the compiler generates SIMD `mov` instruction, while regular `mov` instructions are used for variable-length. The *get* performance is also lower for variable records, as they additionally require more data reads than fixed records. Viper must read the size metadata before retrieving the actual value, while fixed records require only pointer arithmetic due to known offsets at compile time.

5.3.4 Memory Consumption. We evaluate the total DRAM and PMem consumption to better understand the systems' resource requirements. We fill each system with the default 100 million records, i.e., 20.1 GB raw data (1GB = 2^{30}B). We measure the DRAM and PMem consumption with Intel's *VTune*[4], `pmap`[5], and `pmempool`[6].

The results are shown in Figure 9. Viper consumes 21.2 GB of PMem and 2.3 GB of DRAM. The DRAM consumption is attributed nearly completely to the offset map. FASTER consumes slightly less memory overall but significantly more DRAM due to its volatile log, which holds a large part of the data. Dash and µTree both require 23.8 GB for the data via the allocator, being ~10% less efficient than Viper. However, Dash requires only an additional 2.1 GB PMem for its index while µTree requires nearly 9 GB of DRAM for its tree index. pmemkv is very inefficient in its memory consumption, requiring more than twice the raw data size in PMem at 52 GB. In our implementation, CrlStore requires 28 GB of DRAM and 41 GB of PMem, as it needs to store each record twice.

DRAM is a scarce and expensive resource compared to PMem, with a capacity of only about 1/8x on our server and a 9x higher $/GB ratio [1, 18]. Viper's DRAM-PMem ratio is ~1/10 for 216 B records and lower for larger keys due to fingerprinting, i.e., the DRAM consumption depends solely on the number of records. Thus, Viper efficiently manages DRAM and supports larger configurations.

5.3.5 Update Strategy. A recent study by Facebook shows that certain workloads consist of many small updates, e.g., 8 B counter updates [3]. For these workloads, efficient in-place modification significantly reduces read- and write-amplification. In Figure 10 we show the advantage of in-place updates over copy-on-write (*CoW*) updates. When atomically updating only 8 B of a value, Viper achieves more than 2x updates/s compared to CoW. If an atomic update is not possible, Viper still outperforms the other systems when reading, modifying, and re-inserting the value (cf. Fig 6). Recent work [23, 51] has also shown the advantage of in-place updates, thus, supporting larger in-place modifications poses an interesting challenge for future work.

5.3.6 Viper Versions. In Figure 11, we evaluate four Viper versions to understand the impact of data placement in our design, i.e., by placing data + index in *PMem* or *DRAM*, by placing the data in PMem and the index in DRAM (*Viper)*, and by using unaligned VPages in PMem shifted by 2048 Byte (*Unaligned).* We run the experiments with 36 threads. This evaluation supports our design choice of DIMM-aligned storage, as unaligned writes reduce *put* performance by 11%, due to a worse thread-to-DIMM distribution. Random *gets* are affected less than 1%, as they are point lookups that rarely cross DIMM borders.

Placing all data in PMem achieves only ~1/3x performance of the hybrid approach, clearly showing the advantage of a hybrid design when aiming for higher throughput. In Figure 12, we see that 60% of a *put* are already spent in PMem. Adding the index to PMem increases the absolute time spent on *Offset Map* operations and decreases PMem bandwidth due to inefficient access.

On the other side, hybrid Viper achieves ~2/3x of a DRAM-only Viper. The 1.4 µs spent in PMem for *put* are now approximately halved (cf. Fig 1a), reducing the *put* duration to ~1.6 µs, allowing for ~22 Mops/s. Similarly, the time spent fetching data from PMem is reduced by 2.5x, allowing for ~50% more ops/s. This evaluation shows that a hybrid approach significantly outperforms a PMem-only one, while the cost of data persistence is only about 33%. To further close the gap between PMem- and DRAM-based storage, we plan to investigate caching strategies in DRAM in future work.

5.3.7 Operation Breakdown. To better understand the individual operations, we break them down into common sub-parts. We prefill

[4]https://software.intel.com/content/www/us/en/develop/tools/vtune-profiler.html
[5]https://linux.die.net/man/1/pmap
[6]https://pmem.io/pmdk/manpages/linux/v1.8/pmempool/pmempool-info.1.html

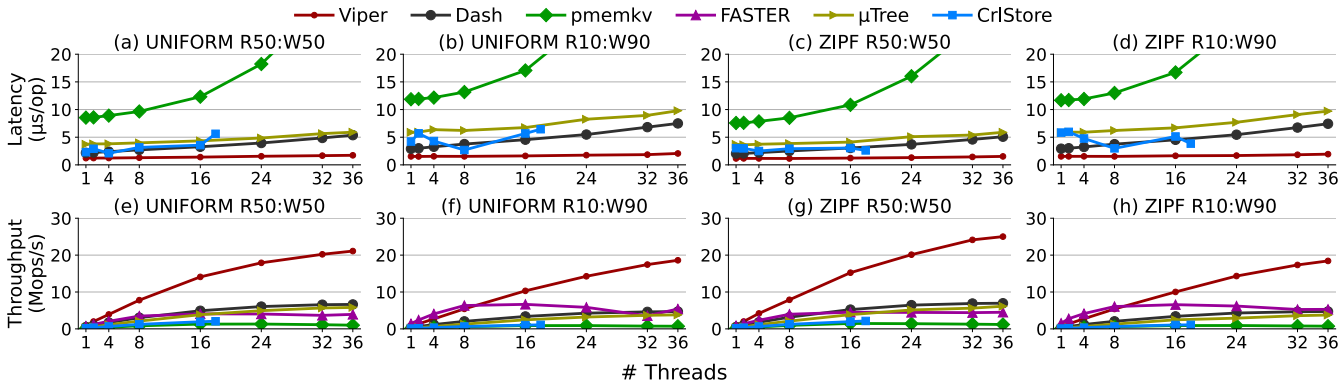


Figure 13: YCSB latency and throughput.

Viper before performing 50 million operations with 36 threads. We split the operations evenly into a mixed 25%-each workload. We normalize the runtime of each operation to 1 and present the time spent on PMem access, Offset Map access, and VPage fetching/locking.

The results are shown in Figure 12. For *put*, we see that most of the time (~60%) is spent on writing the record to PMem. Due to its VPage and client design, very little time is spent on locking and fetching the VPage, as it is cached in the client. As PMem-write speeds are close to those of DRAM, Viper makes good use of the time spent on inserting. However, adding random PMem writes, e.g., in a persistent index, might significantly impact the performance benefits gained by the sequential VPage writes.

Both *updates* and *deletes* require ~30% of the operation time to initially fetch the required VPage and lock it. The majority of the time is then spent in the map, which also includes fingerprint lookups in PMem, to retrieve the correct record offset. The final record update/invalidation is only a small part of the operation.

Retrieving data is similar to updates and deletes, in that it initially requires ~20% to fetch the VPage (but not lock it). Again, the majority (> 50%) is spent in the map lookup and fingerprint resolution. Finally, compared to updates/deletes, 20% is spent on retrieving the actual record and copying it to DRAM.

This breakdown shows that Viper efficiently handles the core operations. The DRAM-based index access takes up a significant portion and Viper might benefit from different index designs, which we plan to investigate in future work.

5.3.8 Space Reclamation. To evaluate the impact of space reclamation on insert and get workloads, we prefill Viper with 216 Byte records and randomly delete 33% of the records without space reclamation. We then manually trigger a compaction of all VBlocks and start 32 parallel threads that *put* or *get* records. Our evaluation shows that running space reclamation in the background has only a marginal impact on the performance of read workloads, i.e., ~2% and no impact on write workloads, as each client inserts records independently and Viper reuses existing VBlocks without new allocations. Thus, space reclamation should be used to reduce the PMem footprint if free CPU resources are available. If Viper is not run at capacity, reclamation can be parallelized to reduce its runtime or a higher threshold can be set to avoid reclaiming every deleted record.

5.3.9 Recovery. As a persistent KVS should be able to restart after a crash or shutdown, we evaluate Viper's recovery performance. We prefill 100 million 216 Byte records and recover using a varying number of threads. A single thread requires 38 seconds to fully restart Viper. More threads reduce the recovery time to 19/10/5/4 seconds with 2/4/8/16 threads. 36 threads recover Viper in 2.3 s.

A disadvantage of a hybrid KVS is that the volatile index needs to be rebuilt when restarting. For a very large KVS, e.g., 1 TB, this can take up to 2 minutes. In Viper, we optimize for the average case of a running database, i.e., improve *put*/*get* performance instead of the worst case, i.e., a crash. However, recovery time is an important aspect of KVSs and we plan to investigate the trade-off between operational and recovery performance in future work.

5.4 YCSB

In this section, we evaluate Viper and the other systems with the widely used Yahoo Cloud Serving Benchmark (YCSB) [8]. We discuss latency and throughput as both are important metrics depending on the exact application, as well as mixed workloads. We split our evaluation along three axes, *i)* latency and throughput (top/bottom row), *ii)* uniform and Zipfian distribution (left/right half), and *iii)* 50:50 and 10:90 read:write workloads (left/right quarter). As YCSB is Java-based and Viper does not offer a network interface, we generate the workloads (8 B keys, 200 B values) using YCSB and then map them into our C++ benchmark for execution. We show the average latency in microseconds measured with *HdrHistogram* [15] and the throughput in million operations/s.

The results are shown in Figure 13. We first look at the latency measurements in the top row, i.e., (a) – (d). We see that Viper has a very low average latency for all four workloads. It increases from 1.2 µs with one thread to a maximum of 2 µs with 36 threads. The Zipfian workloads show a slightly lower latency, due to better caching effects. Dash and µTree have similar latency for all workloads, which is 3–5x higher than Viper's and is mainly caused by the random record allocation. CrlStore also shows low latency, as writes return as soon as they are persisted in the log and frontend KVS. However, while the average latency is low, the 99.9^{th}-percentiles of Dash/µTree/CrlStore in the uniform workloads reach 150/110/240 µs compared to only 25 µs in Viper. pmemkv has significantly higher latency than the other systems in all workloads and peaks at ~50 µs. For all systems, we see a slightly lower latency in the 50:50

workloads compared to the write-heavy workloads as *get* requests perform better in all systems. As FASTER is inherently asynchronous and request completion intervals must be tuned by the user, we omit its latency as it is not directly comparable.

The throughput of all systems follows the trend of the respective latency. For Viper, we see slightly lower maximum throughput (~20 Mops/s) in the uniform workloads than expected compared to the average of the individual *put* and *get* operations as shown in Figure 6, which would reach ~24 Mops/s. In the realistic YCSB workload, there is more mixed access to PMem, which decreases the bandwidth [54], compared to our isolated micro benchmarks. The throughput of the other systems is also similar to the numbers shown in Figure 6. However, all systems are severely limited by inefficient insert operations. Dash and µTree peak at ~8 Mops/s and the other systems reach fewer than 5 Mops/s.

YCSB shows that Viper consistently outperforms existing KVSs with an average latency below 2 µs/op and a maximum throughput of over 19 Mops/s for both write-heavy and mixed workloads. Overall, Viper's throughput is significantly higher in all workloads compared to the other systems, ranging from 3x to 27x, making its design choices a good fit for real-world workloads.

6 RELATED WORK

Viper builds on many techniques from prior KVSs, concurrent hash maps, pure PMem data structures, and hybrid PMem-DRAM structures. In this section, we briefly discuss related work.

Traditional Key-Value Stores. Popular in-memory KVSs such as Redis [45], memcached [35], or MICA [26] optimize for a purely in-memory cache-like use case for maximum performance. They do not persist the data in order to avoid expensive disk access at the cost of data-loss after a system shutdown or crash.

Prior research in persistent KVSs is extensive and focuses mainly on avoiding expensive read- and write-amplification to either SSDs, HDDs, or both [31, 36, 47]. Popular stores such as RocksDB [11], LevelDB [13], and Cassandra [21] use log-structured merge trees with an in-memory table for insertions to reduce write-amplification. To ensure the persistence of the data in the in-memory table, they often employ file-based Write-Ahead-Logging (WAL), which quickly becomes a bottleneck. *FASTER* [5] is a modern KVS that uses an in-memory hash index and a hybrid log to store records on disk with a volatile "tail" that allows for in-place updates. Data in the volatile tail may be lost during a crash. While this approach works very well in some use-cases, we aim for a stronger storage model in Viper, in which data-persistence is guaranteed. With Viper, we propose a persistent KVS that that leverages PMem instead of disk to allow for efficient operation without central log-based bottlenecks.

PMem-Based Key-Value Stores. Recent research also focuses on PMem-based KVSs. *RStore* [24] is a hybrid PMem-DRAM KVS that focuses on reducing tail-latency via asynchronous message passing and log-structured storage. FlatStore [7] also employs a hybrid design based on record batching and cross-core stealing from RDMA-connected request buffers. As RStore and FlatStore are designed as a KVS server, their core design decisions are tightly coupled to networking, include controlling their own threads, and reducing network overhead through user-space networking. Viper's design as an embedded KVS is significantly different, as it does not require any network interaction and more importantly, it does not control its own threads. *HiKV* [53] proposes a hybrid index for a KVS, where a hash index is stored in PMem and a B-Tree is located in DRAM for efficient range queries. However, as only the B-Tree is located in DRAM, all point queries are performed on PMem essentially making it a PMem-only KVS compared to Viper. *LibreKV* [27] also builds a hybrid index where data is initially inserted into a DRAM-based hash map and later merged into a PMem-based hash map once it reaches a certain filling degree. However, LibreKV does not offer consistency as all data in DRAM is lost during a crash. *NVLevel* [29] is an LSM tree-based KVS that uses multiple PMem-based memtables and compacts these into SSTables on disk once they are full. NVLevel uses disk as its storage medium, thus being limited similarly to other disk-based KVSs. In Viper, we propose PMem-specific access patterns for real hardware to efficiently store and retrieve data directly in and from PMem.

PMem Data Structures. Several (hybrid) PMem data structures have been proposed that introduce concepts used in Viper. *Dash* [30], *NVTree* [55], and *FPTree* [40] use a lock-per-node approach in their hash map and B-Tree structures, which we leverage in our VPages. Various work has focused on the advantages of using a hybrid DRAM-PMem approach [6, 28, 40, 49, 57], from which we derive our hybrid index-storage model. Lersch et al. [23] show that in-place updates are preferred over copy-on-write for PMem and that fingerprinting is an effective mean to reduce PMem lookups.

PMem Programming. Yang et al. [54], Izraelevitz et al. [20], and van Renen et al. [50] show how access patterns affect the performance of PMem, which we rely on in Viper. *PMDK* [42] is the de-facto standard toolkit to interact with persistent memory. We make use of its low-level methods in our implementation.

7 CONCLUSION

In this paper, we present *Viper*, a hybrid PMem-DRAM key-value store that leverages PMem-specific access patterns to efficiently store and retrieve data while providing full data persistence. We propose three key design choices for hybrid PMem-DRAM systems based on efficient PMem access patterns for real hardware, *direct PMem-writes*, *uniform thread-to-DIMM distribution*, and *DIMM-aligned storage segments*. We also discuss how to implement core KVS operations in such a system with regard to correct persistence guarantees. Our evaluation shows the efficiency of our design choices, as Viper significantly outperforms existing PMem-only, hybrid, and disk-based KVSs by 4–18x for write workloads, while matching or surpassing their *get* performance. In future work, we plan to investigate alternative index designs and as PMem shows similar performance characteristics to DRAM for certain access, we want to investigate moving parts of the index to PMem. With Viper, we provide a foundation for future work on PMem-aware storage systems and hybrid PMem-DRAM designs based on real PMem hardware characteristics.

ACKNOWLEDGMENTS

This work was partially funded by the German Ministry for Education and Research (ref. 01IS18025A and ref. 01IS18037A), the German Research Foundation (ref. 414984028), and the European Union's Horizon 2020 research and innovation programme (ref. 957407).

REFERENCES

[1] Paul Alcorn. 2020. Intel Optane DIMM Pricing. https://www.tomshardware.com/news/intel-optane-dimm-pricing-performance,39007.html.

[2] Joy Arulraj, Andrew Pavlo, and Subramanya R. Dulloor. 2015. Let's talk about storage & recovery methods for non-volatile memory database systems. In *SIGMOD*. ACM, 707–722.

[3] Zhichao Cao, Siying Dong, Sagar Vemuri, and David H C Du. 2020. Characterizing, Modeling, and Benchmarking RocksDB Key-Value Workloads at Facebook. In *FAST '20*. USENIX Association, 209–223.

[4] Paris Carbone, Stephan Ewen, Seif Haridi, Asterios Katsifodimos, Volker Markl, and Kostas Tzoumas. 2015. Apache Flink(TM): Stream and Batch Processing in a Single Engine. *IEEE Data Eng. Bull.* 38, 4, 28–38.

[5] Badrish Chandramouli, Guna Prasaad, Donald Kossmann, Justin Levandoski, James Hunter, and Mike Barnett. 2018. FASTER: A Concurrent Key-Value Store with In-Place Updates. In *SIGMOD '18*. ACM, 275–290.

[6] Youmin Chen, Youyou Lu, Kedong Fang, Qing Wang, and Jiwu Shu. 2020. uTree: a persistent B+-tree with low tail latency. *Proceedings of the VLDB Endowment* 13, 12, 2634–2648.

[7] Youmin Chen, Youyou Lu, Fan Yang, Qing Wang, Yang Wang, and 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*. ACM, 1077–1091.

[8] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *SoCC '10*. ACM, 143–154.

[9] Björn Daase, Lars Jonas Bollmeier, Lawrence Benson, and Tilmann Rabl. 2021. Maximizing persistent memory bandwidth utilization for OLAP workloads. In *SIGMOD '21*. ACM.

[10] Benoit Dageville, Jiansheng Huang, Allison W. Lee, Ashish Motivala, Abdul Q. Munir, Steven Pelley, Peter Povinec, Greg Rahn, Spyridon Triantafyllis, Philipp Unterbrunner, Thierry Cruanes, Marcin Zukowski, Vadim Antonov, Artin Avanes, Jon Bock, Jonathan Claybaugh, Daniel Engovatov, and Martin Hentschel. 2016. The Snowflake Elastic Data Warehouse. In *SIGMOD '16*. ACM, 215–226.

[11] Facebook. 2020. RocksDB. https://rocksdb.org.

[12] Apache Flink. 2020. Improvement in (de)serialization of keys and values for RocksDB state. https://issues.apache.org/jira/browse/FLINK-9702.

[13] Google. 2020. LevelDB, a fast key-value storage library. https://code.google.com/p/leveldb.

[14] Philipp Götze, Arun Kumar Tharanatha, and Kai-Uwe Sattler. 2020. Data structure primitives on persistent memory: an evaluation. In *DaMoN '20*. ACM, 14:1–14:3.

[15] HdrHistogram. 2020. HdrHistogram: A high dynamic range histogram. http://hdrhistogram.org.

[16] Daokun Hu, Zhiwen Chen, Jianbing Wu, Jianhua Sun, and Hao Chen. 2021. Persistent Memory Hash Indexes: An Experimental Evaluation. *Proceedings of the VLDB Endowment* 14, 5, 785–798.

[17] 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. 967–979.

[18] Intel. 2020. Intel® Optane™ Persistent Memory. https://intel.com/optanedcpersistentmemory.

[19] Intel. 2020. TBB concurrent hash map. https://software.intel.com/en-us/node/506077.

[20] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R. Dulloor, Jishen Zhao, and Steven Swanson. 2019. Basic Performance Measurements of the Intel Optane DC Persistent Memory Module. *arXiv:1903.05714 [cs]*.

[21] Avinash Lakshman and Prashant Malik. 2010. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review* 44, 2, 35–40.

[22] Per-Ake Larson. 1988. Dynamic hash tables. *Commun. ACM* 31, 4, 446–457.

[23] Lucas Lersch, Xiangpeng Hao, Ismail Oukid, Tianzheng Wang, and Thomas Willhalm. 2019. Evaluating persistent memory range indexes. *Proceedings of the VLDB Endowment* 13, 4, 574–587.

[24] Lucas Lersch, Ivan Schreter, Ismail Oukid, and Wolfgang Lehner. 2020. Enabling low tail latency on multicore key-value stores. *Proceedings of the VLDB Endowment* 13, 7, 1091–1104.

[25] Xiaozhou Li, David G. Andersen, Michael Kaminsky, and Michael J. Freedman. 2014. Algorithmic improvements for fast concurrent Cuckoo hashing. In *EuroSys '14*. ACM, 1–14.

[26] Hyeontaek Lim, Dongsu Han, David G. Andersen, and Michael Kaminsky. 2014. MICA: a holistic approach to fast in-memory key-value storage. In *NSDI' 14*. USENIX Association, 429–444.

[27] Hao Liu, Linpeng Huang, Yanmin Zhu, and Yanyan Shen. 2017. LibreKV: A Persistent In-Memory Key-Value Store. *IEEE Transactions on Emerging Topics in Computing*, 1–1.

[28] Jihang Liu, Shimin Chen, and Lujun Wang. 2020. LB+Trees: optimizing persistent index performance on 3DXPoint memory. *Proceedings of the VLDB Endowment* 13, 7, 1078–1090.

[29] Ruicheng Liu, Peiquan Jin, Xiaoliang Wang, Zhou Zhang, Shouhong Wan, and Bei Hua. 2019. NVLevel: A High Performance Key-Value Store for Non-Volatile Memory. In *HPCC/SmartCity/DSS '19*. IEEE, 1020–1027.

[30] Baotong Lu, Xiangpeng Hao, Tianzheng Wang, and Eric Lo. 2020. Dash: scalable hashing on persistent memory. *Proceedings of the VLDB Endowment* 13, 8, 1147–1161.

[31] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Hariharan Gopalakrishnan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2017. WiscKey: Separating Keys from Values in SSD-Conscious Storage. *ACM Transactions on Storage* 13, 1, 5:1–5:28.

[32] Kelly Lyon. 2021. How Intel Optimized RocksDB Code for Persistent Memory with PMDK. https://software.intel.com/content/www/us/en/develop/articles/how-intel-optimized-rocksdb-code-for-persistent-memory-with-pmdk.html.

[33] Kazuaki Maeda. 2012. Performance evaluation of object serialization libraries in XML, JSON and binary formats. In *DICTAP '12*. IEEE, 177–182.

[34] Anton Malakhov. 2015. Per-bucket concurrent rehashing algorithms. *arXiv:1509.02235 [cs]*.

[35] Memchached. 2020. Memcached, high-performance, distributed memory object caching system. https://https://memcached.org/.

[36] Prashanth Menon, Tilmann Rabl, Mohammad Sadoghi, and Hans-Arno Jacobsen. 2014. CaSSanDra: An SSD boosted key-value store. In *ICDE '14*. IEEE, 1162–1167.

[37] mmap. 2020. mmap(2): map/unmap files/devices into memory - Linux man page. https://linux.die.net/man/2/mmap.

[38] Moohyeon Nam, Hokeun Cha, Young-Ri Choi, Sam H. Noh, and Beomseok Nam. 2019. Write-optimized dynamic hashing for persistent memory. In *FAST '19*. USENIX Association, 31–44.

[39] Faisal Nawab, Joseph Izraelevitz, Terence Kelly, Charles B. Morrey III, Dhruva R. Chakrabarti, and Michael L. Scott. 2017. Dalí: A Periodically Persistent Hash Map. In *DISC '17*. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 37:1–37:16.

[40] 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 *SIGMOD '16*. ACM, 371–386.

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

[42] PMDK. 2020. Persistent memory programming. https://pmem.io/pmdk.

[43] PmemKV. 2020. pmemkv, key/value datastore for persistent memory. https://pmem.io/pmemkv.

[44] PmemObj++. 2020. libpmemobj++ concurrent hash map. https://github.com/pmem/libpmemobj-cpp.

[45] Redis. 2020. Redis, an in-memory data structure store. https://redis.io.

[46] David Schwalb, Markus Dreseler, Matthias Uflacker, and Hasso Plattner. 2015. NVC-Hashmap: A Persistent and Concurrent Hashmap For Non-Volatile Memories. In *IMDM '15*. ACM, 4:1–4:8.

[47] Pradeep Shetty, Richard Spillane, Ravikant Malpani, Binesh Andrews, Justin Seyster, and Erez Zadok. 2013. Building workload-independent storage with VT-trees. In *FAST '13*. USENIX Association, 17–30.

[48] Julian Shun and Guy E. Blelloch. 2014. Phase-concurrent hash tables for determinism. In *SPAA '14*. ACM, 96–107.

[49] Alexander van Renen, Viktor Leis, Alfons Kemper, Thomas Neumann, Takushi Hashida, Kazuichi Oe, Yoshiyasu Doi, Lilian Harada, and Mitsuru Sato. 2018. Managing Non-Volatile Memory in Database Systems. In *SIGMOD '18*. ACM, 1541–1555.

[50] Alexander Van Renen, Lukas Vogel, Viktor Leis, Thomas Neumann, and Alfons Kemper. 2019. Persistent memory I/O primitives. In *DaMoN '19*. ACM, 12:1–12:7.

[51] Alexander van Renen, Lukas Vogel, Viktor Leis, Thomas Neumann, and Alfons Kemper. 2020. Building blocks for persistent memory. *The VLDB Journal*.

[52] Tianzheng Wang, Justin Levandoski, and Per-Ake Larson. 2018. Easy Lock-Free Indexing in Non-Volatile Memory. In *ICDE '18*. IEEE, 461–472.

[53] Fei Xia, Dejun Jiang, Jin Xiong, and Ninghui Sun. 2017. HiKV: a hybrid index key-value store for DRAM-NVM memory systems. In *ATC '17*. USENIX Association, 349–362.

[54] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steven Swanson. 2020. An Empirical Guide to the Behavior and Use of Scalable Persistent Memory. In *FAST '20*. USENIX Association, 169–182.

[55] 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 '15*. USENIX Association, 167–181.

[56] Matei Zaharia, Scott Shenker, Haoyuan Li, Tathagata Das, Timothy Hunter, and Ion Stoica. 2013. Discretized streams: fault-tolerant streaming computation at scale. In *SOSP '13*. ACM, 423–438.

[57] Xinjing Zhou, Lidan Shou, Ke Chen, Wei Hu, and Gang Chen. 2019. DPTree: differential indexing for persistent memory. *Proceedings of the VLDB Endowment* 13, 4, 421–434.