p^{2}KVS: a Portable 2-Dimensional Parallelizing Framework to Improve Scalability of Key-value Stores on Solid-State Drives

Anonymous Author(s)
Submission Id: 151

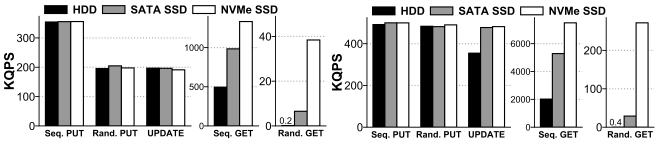
Abstract

Attempts to improve the performance of key-value stores (KVS) by replacing the slow Hard Disk Drives (HDDs) with much faster Solid-State Drives (SSDs) have consistently fallen short of the performance gains implied by the large speed gap between SSDs and HDDs, especially for small KV items. We experimentally and holistically explore the root causes of performance inefficiency of existing LSM-tree based KVSs running on powerful modern hardware with multicore processors and fast SSDs. Our findings reveal that the global write-ahead-logging (WAL) and index-updating (MemTable) can become bottlenecks that are as fundamental and severe as the commonly known LSM-tree compaction bottleneck, under both the single-threaded and multi-threaded execution environments.

To fully exploit the performance potentials of full-fledged KVS and the underlying high-performance hardware, we propose a portable 2-dimensional KVS parallelizing framework, referred to as p^{2}KVS. In the horizontal inter-KVS-instance dimension, p^{2}KVS partitions a global KV space into a set of independent subspaces, each of which is maintained by an LSM-tree instance and a dedicated worker thread pinned to a dedicated core, thus eliminating structural competition on shared data structures. In the vertical intra-KVS-instance dimension, p^{2}KVS separates user threads from KVS-workers and presents a runtime queue-based opportunistic batch mechanism (OBM) on each worker, thus boosting process efficiency. Since p^{2}KVS is designed and implemented as a user-space request scheduler, viewing WAL, MemTables, and LSM-trees as black boxes, it is nonintrusive and highly portable. Under micro and macro-benchmarks, p^{2}KVS is shown to gain up to 4.6× write and 5.4× read speedups over the state-of-the-art RocksDB.

1 Introduction

Key-value Store has become a building block of modern IT infrastructure to support a myriad of upper-layer applications [13, 33, 41]. Most current production KVSs, such as RocksDB [21], LevelDB [23], HBase [2], and Cassandra [34], adopt log-structured merge trees (LSM-trees) [43] to batch random writes in memory, and then flush them to storage sequentially, which is IO friendly for traditional Hard Disk Drives (HDDs). Furthermore, LSM-tree-based indexing structures keep KV pairs sorted in storage, thus accelerating retrieves and scans.



(a) Single user thread

(b) 8 user threads

Figure 1. RocksDB single-threaded and multi-threaded throughput performance. *HDD: WDC WD100EFAX 10TB, SATA SSD: Samsung 860 PRO 512 GB, NVMe SSD:Intel Optane 905p 480 GB. The hardware platform is equipped with two 22-core CPUs. The KV size is 128-byte.*

Modern computer systems with high-performance Solid-State Drives (SSDs) and multicore processors are expected to dramatically improve the overall performance of KVS over the older HDD-based systems. Unfortunately, simply replacing slow HDDs with fast SSDs has failed to consistently deliver the write performance of KVS promised by SSD's with 10× higher IO bandwidth and 100× higher IOPS than HDDs, especially for small key-value pairs. Figure 1a illustrates that although RocksDB achieves up to 2 orders of magnitude higher read performance from superior SSDs, its write performance remains almost unchanged between SSDs and HDDs. Recent research [19, 35] and our experimental results consistently suggest that *write workloads with small-sized KV pairs overload a system's host CPU-cores rather than being bottlenecked by the system's IO bandwidth.* Nevertheless, a naïve approach to increasing the processing capacity is to fully leverage the power of the multicore CPU by invoking more user threads. Figure 1b shows that even with 8 user threads the query-per-second (QPS) performance under the write workloads for sequential PUT, random PUT, and UPDATE, is only improved by about 40%, 150%, and 160%, respectively, far short of the desired linear scaling. However, compared to the single-threaded cases, the performance of the three 8-threaded cases under write workloads improve from HDD-based system to SSD-based system by less than 10%, except for the update operation case with a 40% gain.

Generally, foreground user threads each first execute WAL and then execute indexing while its background threads execute compactions. Previous studies have reported the potential bottlenecks in the logging[10], indexing[5], and compaction[4, 8, 40] stages separately. To alleviate the WAL bottleneck, RocksDB employs an async-logging approach at the risk of data loss after a crash. Furthermore, RocksDB and SpanDB [12] batch concurrent writes to fewer-but-larger ones but at the expense of extra synchronization overhead. To mitigate the MemTable index-updating bottleneck, prior works propose novel concurrent data structures to provide lock-free, write-friendly, or highly-concurrent functionalities [5, 16, 17, 54], but only gaining limited speedups (e.g., up to 2×) at the added cost of design and implementation complexity. Great efforts have been made to optimize compaction, such as leveraging fast NVM [24, 25, 29, 30, 52], parallelizing the work process [14, 35], size-tiered [34, 45, 48], and key-and-value separation [9, 40]. To exploit potentials of modern hardware with high parallelism, KVell [35] and SplinterDB [14] propose new data structures in lieu of LSM-tree to expose IO and CPU concurrency and mitigate write amplification. The production-level KVSs (e.g., RocksDB) have been evolving with hardware advances by developing optimization techniques, such as batch-write, concurrent skiplist, multi-threaded compaction, etc. The KVS sharding mechanism with multiple instances is widely used in common database practices to exploit the inter-instance parallelism [2, 19, 20, 22, 27, 53]. However, existing works still cannot comprehensively explain and effectively address the poor performance scalability with high-performance hardware.

To help understand the root causes of the performance inefficiency of mature KVSs on fast SSDs and powerful processors, we conduct a series of experiments with the well-optimized RocksDB (detailed in Section 3). The results provide three revealing findings. First, with a single user thread, either logging or indexing can impose a serious compute bottleneck that severely limits the performance of random small-sized writes. Only when both logging and indexing are no longer the bottlenecks, will LSM-tree compaction become a storage bottleneck. Second, increasing the number of user threads yields marginal benefits because of the intensive contentions on the shared log and index structures, the more the user threads, the worse the contentions. Third, the KVS sharding mechanism with multiple instances still suffers from the contention from multiple user threads, as well as inefficient logging and indexing.

To overcome the architectural drawback of existing KVS systems, we propose a portable 2-dimensional parallelizing KVS framework, referred to as p^{2}KVS, to effectively leverage both matured production KVS implementations (e.g., RocksDB) and their underlying high-performance hardware. First, in the horizontal dimension of inter-KVS-instance parallelism, p^{2}KVS adopts a scheduling scheme for multiple KVS-worker threads that are each pinned to dedicated individual cores. Each worker maintains its own independent WAL log, MemTable, and LSM-tree, thus reducing the contention on shared data structures. Second, in the vertical dimension of intra-KVS-instance parallelism, p^{2}KVS designs a global KV accessing layer to separate user threads from KVS-worker threads. The accessing layer strategically distributes all incoming requests to the workers' queues, balancing load among the cores. Third, p^{2}KVS presents a runtime queue-based opportunistic batching mechanism (OBM) in each worker. For outstanding write requests in the queue, OBM merges them to amortize the overheads of both KV-handling and logging. For outstanding read requests, OBM invokes the existing multiget function to improve processing efficiency. Different from multi-instance KVS configurations, p^{2}KVS explicitly eliminates the potential contention among user threads upon an instance while simultaneously leveraging a batching mechanism to improve efficiency.

The goal of p^{2}KVS is to design an efficient thread-based parallelizing framework upon RocksDB or other existing KVSs, to improve the processing capacity by fully leveraging modern hardware characteristics while leaving their existing features and internal designs intact. Therefore, p^{2}KVS is portable and nonintrusive.

The contributions of p^{2}KVS are summarized as follows:

- We experimentally and holistically analyze and identify the root causes of poor scalability experienced by KVSs running on fast SSDs and multicore processors.
- We present p^{2}KVS, a portable 2-dimensional KVS parallelizing framework to uniformly and effectively exploit internal parallelism among and within KVS instances. p^{2}KVS separates user threads from worker threads and distributes requests among the worker threads. It designs a queue-based opportunistic batching mechanism to improve the handling efficiency of each worker.
- We implement p^{2}KVS prototypes on RocksDB and LevelDB, to evaluate it through extensive experiments under popular macro- and micro-benchmarks. Compared with the state-of-the-art LSM-tree based RocksDB and PebblesDB, p^{2}KVS achieves up to 4.6× write and 5.4× read speedups for small-sized KVs. It also outperforms the state-of-the-art Non-LSM-tree based KVell.

The rest of this paper is organized as follows. Section 2 introduces LSM-tree based KVS and multi-threaded optimizations of RocksDB on SSDs. Section 3 analyzes the root causes of the observed poor scalability of running RocksDB on modern hardware. Section 4 presents the design of p^{2}KVS. We evaluate p^{2}KVS in Section 5, discuss related works in Section 6 and conclude the paper in Section 7.

2 Background

2.1 LSM-tree based Key Value Store

Log-Structured Merge Tree (LSM-tree) based KVS were originally designed for write-intensive workloads on HDDs. As shown in Figure 2, a LSM-tree based KVS consists of three main components: log, MemTables, and on-disk LSM-Tree. During write operations, key-value (KV) pairs are written to these three components in sequence. A KVS implementation

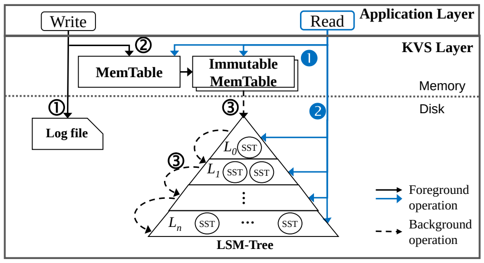


Figure 2. The main components of LSM-tree based KVS and the processing flow of KV write and read requests. ①*Write-ahead logging.* ②*MemTable Index updating.* ③*LSM-tree compaction.* ❶*Search in MemTables.* ❷*Search in LSM-tree.*

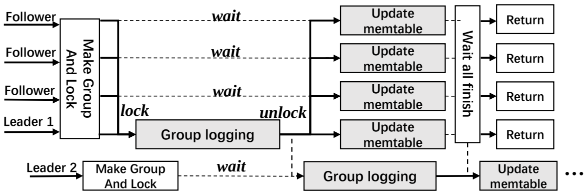


Figure 3. RocksDB concurrent write process.

is generally a user library so that its foreground threads are also user threads.

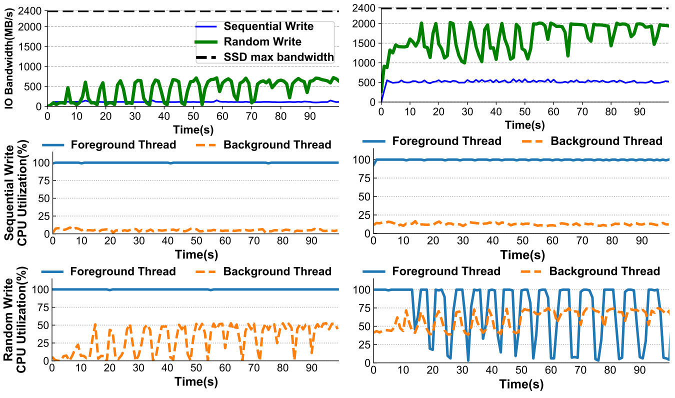
Write. When a user thread from an application writes a KV pair (e.g., PUT, DELETE, and UPDATE), it first writes the KV pair into the log file for fast persistence, a process referred to as *write-ahead-logging* (WAL①). And then, the user thread inserts the KV pair into a *MemTable* buffered in memory and updates the corresponding index structure (e.g., skiplist), a process referred to as *index-updating* (②).

When a MemTable reaches its predefined capacity, it becomes a read-only *Immutable MemTable*, and then is transferred to the LSM-tree as a Sorted-String-Table (SST) by the background thread, a process referred to as *minor compaction*. After that, its corresponding log records will be removed.

The on-disk LSM-tree contains multiple levels (L_0, L_1, L_2, $\ldots L_n$) with exponentially growing capacity. Each level consists of multiple fix-sized SSTs, containing sorted KV pairs and metadata. To ensure that the LSM-tree structure is kept with exponentially growing sorted levels, the background threads of KVS also trigger *major compactions* to merge SSTs from higher levels to lower levels. We refer to both minor and major compaction as LSM-tree compaction (③).

Only when a level in LSM-tree reaches its capacity, is compaction triggered, consuming SSD bandwidth and indirectly impacting the write processing [4, 8, 38].

Read. Read-type requests (e.g., GET and SCAN) search the requested keys in the MemTable and the on-disk LSM-tree. For point query (i.e., GET), KVS first searches in MemTables through the skiplist index (❶). When the key is not found in memory, the LSM-tree on the disk is searched from higher to lower levels until the requested KV is found(❷). For range query (i.e., RANGE and SCAN), KVS scans MemTables and LSM-tree to retrieve all KV pairs in the requested range.



(a) 128-byte KV size **(b)** 1KB KV size

Figure 4. RocksDB IO bandwidth and CPU utilizations. *a user thread continuously inserts 100M KV pairs sequentially and randomly, respectively, on an Optane SSD.*

2.2 RocksDB optimizations for concurrency

As a well-optimized production-level LSM-tree based KVS, RocksDB has implemented many optimizations and configurations to improve its QPS performance by exploiting hardware parallelism. An example of concurrent write process in RocksDB is shown in Figure 3. The key concurrency optimizations are as follows:

Group logging. When multiple user threads submit write requests concurrently, RocksDB organizes them into a group. One of these threads is elected as the leader responsible for aggregating log entries from all threads in the group and then writing the log file once, while the other threads, called followers, are suspended until the log-file write is complete. This reduces the actual log IOs, thus improving IO efficiency.

Concurrent MemTable. RocksDB supports a concurrent skiplist index, improving QPS of MemTable insertions by up to 2× over the vanilla skiplist. Like group logging, a group of user threads that concurrently update MemTable are synchronized when updating the global metadata.

Pipelined write. RocksDB pipelines the logging and index-updating steps in different groups to reduce blocking.

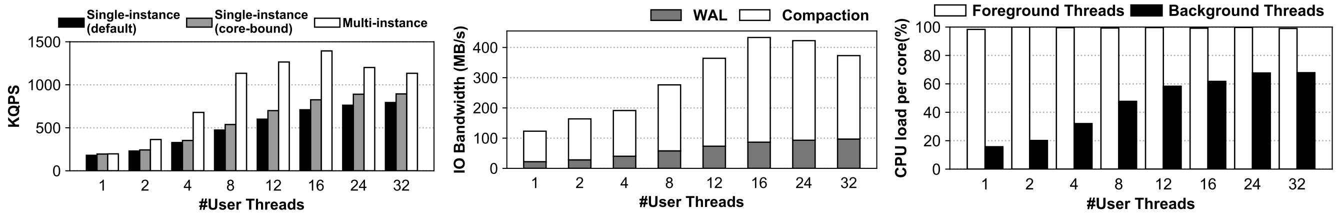
LSM-tree based KVSs such as LevelDB and RocksDB typically provide a request batching operation, called WriteBatch, which allows users to perform multiple write-type requests in a batch. RocksDB merges the logging operations of all the requests in a WriteBatch like the group logging mechanism.

3 Root Causes of Poor Scalability

3.1 Single User Thread

Modern SSDs outperform HDDs in both read and write bandwidth by one order of magnitude (e.g., about 2 GB/s vs. 0.2 GB/s). Furthermore, SSDs with high internal parallelism exhibit almost 2~4 orders of magnitude higher IOPS than HDDs, especially for random and small IOs.

Intuitively, replacing HDDs with superior SSDs should significantly boost the performance of LSM-tree based KVSs.



(a) Throughput **(b)** IO bandwidth of the single-instance case **(c)** Average CPU utilization of the single-instance case

Figure 5. Performance analysis of concurrent writes in RocksDB. *Each experiment performs 10M random writes with 128-byte key-value pairs on an Intel Optane 905p SSD.*

To validate this, we perform 5 sets of 10M 128-Byte KV operations (i.e., sequential and random PUT, random UPDATE, sequential and random GET) on RocksDB with an HDD, a SATA SSD, and an NVMe SSD, respectively. Workloads with small-sized key-value pairs are considered a common case, as it has been reported that 90% of KV pairs in typical RocksDB workloads are less than 1KB and the average key-value size is less than 100 bytes [7]. The results are shown in Figure 1. Although the sequential and random read performances of RocksDB on the SSDs are up to 3× and 200× higher than those on the HDD respectively, the write performances are only comparable to or slightly higher than HDD.

Figure 4a further shows that a user thread inserting small-sized KV pairs over time in the random and sequential cases. The thread with 100% CPU-core utilization consumes only 1/6 and 1/20 of the full SSD IO bandwidth respectively. The continuous random writes trigger periodic flushes and compactions performed by the corresponding background threads that consume about 25% CPU-core utilization. When the user thread with about 70% CPU-core utilization continuously inserts random large-sized KV pairs (i.e., 1KB), only the periodic compactions by the background threads consume 23% IO bandwidth and 60% CPU-core utilization as indicated in Figure 4b. Most previous studies in this area consider LSM-tree compaction as a primary factor severely hampering the overall performance because of its high IO intensity with write stall and write amplification [1, 3, 18, 40, 42, 45, 48, 52]. In fact, *write workloads with small-sized KV pairs overload CPU-cores but underutilize IO bandwidth, while the exact opposite is true for write workloads with large-sized KV pairs.*

3.2 Multiple User Threads

Both multicore-processors and NVMe based SSDs have adequate computing and IO capacity with high parallelism, respectively. Naturally, increasing the number of user threads leveraging powerful hardware can enhance the overall throughput of KVS. Alternatively, in practice, database practitioners also simply configure multiple independent KVS instances on superior hardware to improve the overall performance. In this analysis, we consider the two cases of single-instance and multi-instance that are accessed by multiple user threads. Each KVS instance has its own independent log file, MemTable, and LSM-tree.

The results in Figure 5a show the scalability in both cases. The write QPS in the single-instance case with all parallel optimizations still scales poorly and gains a meager 3× speedup at 32 user threads. Its throughput peaks at 24 threads and further scaling beyond this point shows diminishing returns. Compared to the single-instance case, the multi-instance case achieves 80% higher throughput and better scalability, with its throughput peaking at less than 16 threads/instances.

The experimental results in Figure 5b clearly suggest again that compaction is not the bottleneck of KVSs on SSDs, at least not the main or dominant one, even in the multi-threaded case. At its peak bandwidth consumption at 16 threads, only 1/5 of the SSD IO capacity (400 MB/s of 2 GB/s) is used. Meanwhile, the bandwidth consumed by compactions is no more than 3/4 of the total bandwidth consumed. Similar to the single-user-thread case, the CPU usages of foreground user threads are close to 100% while the background compaction threads consume relatively low CPU utilizations, as shown in Figure 5c.

In addition, Figure 5a also illustrates the performance benefits of binding threads to the physical CPU cores, increasing the write throughput of RocksDB by 10% to 15% because the bound threads do not switch among CPU cores due to the scheduling of the operating system.

Recent studies, KVell [35] and SplinterDB [14], reported similar low IO bandwidth utilizations and believed that the maintenance (i.e., compaction) and index-synchronization stemming from LSM-tree cause the performance bottleneck of RocksDB on fast storage devices.

In light of the large user installation bases of matured production-level KVSs (e.g., RocksDB), we are motivated to explore solutions that address the root causes of the aforementioned performance inefficiency upon modern hardware with multicore CPUs and superior SSDs in a nonintrusive and portable way.

3.3 Breakdown of Processing Time

Next, we explore the root causes of inefficient scalability in LSM-tree based KVS.

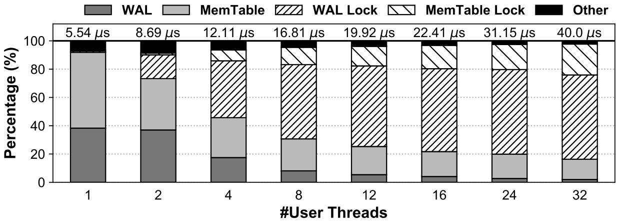


Figure 6. RocksDB write latency breakdown.

Figure 6 shows the latency breakdown of the foreground write threads in the single-instance case of RocksDB. We divide the write process into five steps, WAL, MemTable, WAL lock, MemTable lock, and Others. *WAL* represents the execution time of write-ahead logging, including IO time and the other processes (e.g., encoding log records, calculating checksum, and adding them into a memory buffer). *MemTable* represents the latency of inserting key-value pairs into MemTable, of which more than 90% is updating the skiplist index. *WAL lock* represents the lock overhead associated with the group logging mechanism, including the lock acquisition time of other user threads when a leader thread executes WAL and the time when the leader thread notifies other threads of the completion of the WAL execution. *MemTable lock* indicates the thread synchronization time when the same group of threads concurrently write MemTable. *Others* represents the other software overheads.

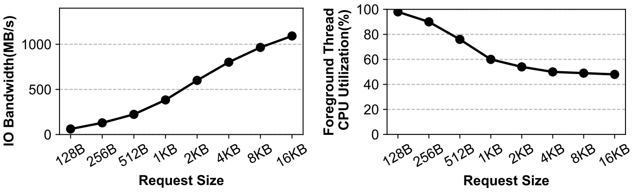
Lock overheads. As the number of writers increases, the combined percentage of CPU cycles for WAL and MemTable decreases from 90% at a single thread to 16.3% at 32 threads, while the total lock overhead (i.e., WAL lock and MemTable lock) increases from almost nothing to 81.4%. More writers introduce heavier contention upon the shared data structures such as the log and MemTable. Particularly, the WAL-lock with just 8 threads accounts for more than half of the latency.

The primary reasons for the high lock overhead are threefold. First, RocksDB's group-logging policy serializes log writes on a leader and suspends the followers; Second, the more threads in the group, the more CPU time is used to unlock the follower threads; Finally, multiple threads inserting into a MemTable introduces the synchronization overhead.

3.4 Multi-threading, a Double-edged Sword

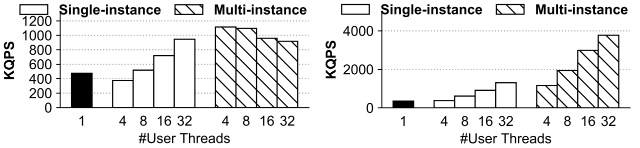
As indicated by experimental results shown above, the advantage of multi-threading in exploiting parallelism can be more than offset by the contentions among the threads on shared data structures such as log and index. Therefore, a careful tradeoff should be found to achieve an optimal overall outcome. With this in mind, next, we investigate the effect of multiple threads on the two key bottlenecks of logging and indexing. We experimentally analyze the performance of singe-instance and multi-instance cases in the WAL process and MemTable process respectively with 128-byte KV workloads, as shown in Figure 8.

Write-ahead-logging. As shown in Figure 6, the average latency of WAL decreases from 2.1 μs at a single user thread to 0.8 μs at 32 user threads. This is because the group logging strategy aggregates small log IOs from different threads into larger IOs, thus improving IO efficiency.



(a) IO bandwidth **(b)** CPU utilization

Figure 7. The effect of write request batching mechanism. *The async-logging approach is enabled.*



(a) Logging **(b)** MemTable

Figure 8. Throughput of different parallelizing schemes on the WAL and MemTable process respectively. *MemTable inserting and indexing are disabled when testing WAL logging, and vice versa.*

To demonstrate the effect of batching mechanism on write performance, we measure the bandwidth and CPU usage of WAL when batching several 128-byte key-value pairs into 256-byte to 16KB sized WriteBatch request, as shown in Figure 7. With the default configuration of RocksDB, the async-logging approach of RocksDB is enabled to eliminate write amplification caused by `fsync` after each log IO. The results show that the request-level batching mechanism can not only improve SSD bandwidth utilization due to the larger IO size but also effectively reduce CPU load due to the reduction of software overhead in the IO stack.

Figure 8a shows that the single-instance case improves QPS by 2× with 32 threads using batching, while the multi-instance case achieves at its peak more than 2.5× QPS with 4 threads. The limited IO parallelism within the underlying SSD largely determines the optimal number of logging threads in the multi-instance case.

Index. Different from the logging process above, the overall throughput in the MemTable index updating process scales well in both single-instance and multi-instance cases, as shown in the Figure 8b, although the latency of updating MemTable increases from 2.9 μs at a single thread to 5.7 μs at 32 threads. Further, the multi-instance case is obviously superior to the single-instance case. Specifically, the throughput gain with the former reaches 10.5× QPS at 32 threads, while the latter only improves QPS by 3.7× at 32 threads in Figure 8b. This performance gap stems primarily from the synchronization overheads and the diminishing return of the shared concurrent skiplist in the latter. This indicates that although the concurrent MemTable allows multiple threads to insert into the skiplist in parallel, its scalability is limited.

In summary, both the single-instance and multi-instance cases have shown their own advantages and disadvantages in scalability when deployed in the current KVS architecture with high-performance hardware. For the WAL logging bottleneck, while the single-instance case can consistently benefit from thread-scaling by leveraging the batching mechanism, the multi-instance case achieves a much higher logging throughput performance but at limited inter-instance parallelism. On the other hand, the overall index-updating performance scales better with the multiple-instance case than with the single-instance case because of the lack of lock contention in the former. Furthermore, since WAL and MemTable are on the same KVS write critical path, with the former preceding the latter in the process flow, the overall throughput is limited by the slower of the two processes. These observations and analysis suggest that a KVS processing architecture that fully harnesses the underlying high-performance hardware should be designed to holistically consider the interplay and tradeoff between the inter- and intra-instance parallelism, between logging and indexing, and between computation and storage overheads.

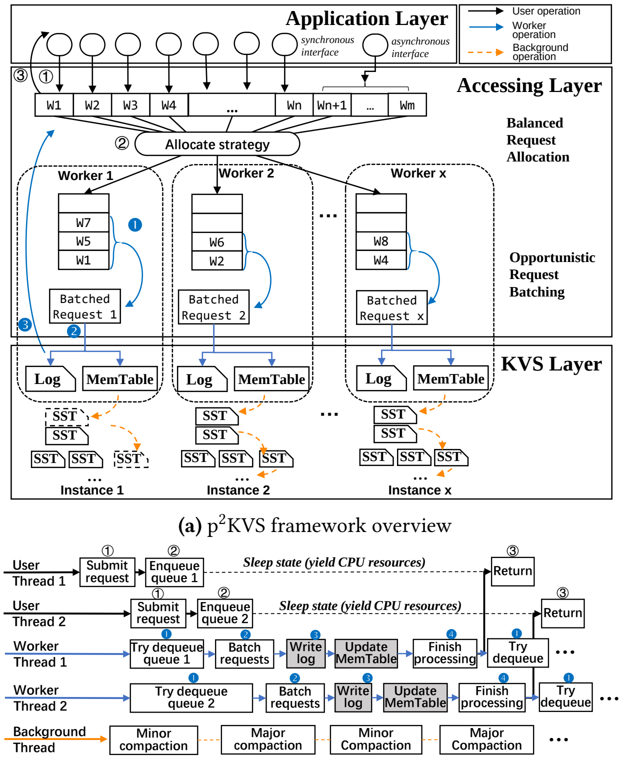
4 Design and Implementation

Our in-depth experimental analysis in the preceding sections motivates us to propose a portable 2-dimensional parallelizing KVS framework, referred to as p^{2}KVS, to effectively leverage both matured production KVS implementations (e.g., RocksDB) and the power of modern hardware. p^{2}KVS takes a three-pronged approach to its design as follows.

- Exploiting inter-instance parallelism with horizontal key space partitioning among multiple KVS instances. p^{2}KVS adopts multiple KVS-worker threads that are each bound to different cores. Each worker runs a KVS instance with its own independent WAL log, MemTable, and LSM-tree, thus avoiding the contention upon shared data structures.
- Exposing intra-instance parallelism with a global KV accessing layer. p^{2}KVS designs a global KV accessing layer to separate user threads from KVS-worker threads. The accessing layer strategically distributes all incoming KV requests from applications to the workers' queues, balancing load among a limited number of workers.
- Alleviating logging and indexing bottlenecks with queue-based opportunistic batching. p^{2}KVS presents a runtime queue-based opportunistic batching mechanism in each worker to amortize the overheads of both KV-handling and logging.

4.1 Overall Architecture

The architecture of p^{2}KVS is shown in Figure 9a. In the vertical dimension, p^{2}KVS adds an accessing layer between the application layer and the KVS layer. The accessing layer accepts user requests from the upper applications. In the horizontal dimension, p^{2}KVS maintains a set of worker threads (workers for short) that each manage a KVS instance and run independently without shared data structures. Each worker has its own request queue and is bound to a CPU core and is responsible for executing requests in its own KVS instance.



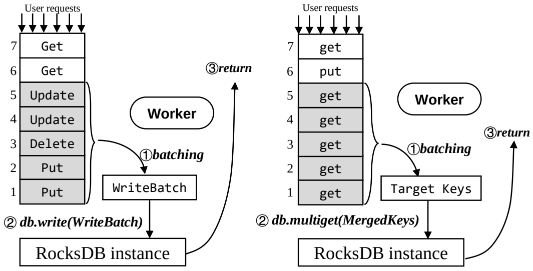
(a) p^{2}KVS framework overview

(b) Workflow of three kinds of threads

Figure 9. Architectural overview and workflow of p^{2}KVS framework. ①Submit request. ②Allocate worker and enqueue. ③Request return. ❶Batch requests from the request queue. ❷Perform processing. ❸Finish processing.

As shown in Figure 9b, in p^{2}KVS, each user thread only submits the requests to the request queue of the corresponding worker according to the allocation strategy (①)(see Section 4.2 for details), and then suspends itself without further CPU consumption (②). The worker handles the enqueued requests in batch (❶) and executes them in the corresponding KVS instance (❷) (see Section 4.3 for details). When a request is handled (❸), its suspended owner user thread will be notified to return (③). Note that the request processing consumes the CPU resources of the worker. Background operations like minor and major compactions are performed by background threads.

p^{2}KVS maintains a global and standard KV interface, e.g., PUT, GET, DELETE, SCAN, etc., and is expected to be totally transparent to the upper applications. However, it redirects KV requests to internally sharded KVS instances, offering inter-instance parallelism. Note that, while databases and applications generally leverage user-specific semantics (e.g.,



(a) Batching write requests **(b)** Batching read requests

Figure 10. Opportunistic batching mechanism on RocksDB.

column family [20]) to assign key-value pairs to the underlying multiple KVS instances, p^{2}KVS provides the standard KV interfaces for the upper application without additional semantics. In addition, p^{2}KVS also extends asynchronous write interfaces (e.g., $Put(K, V, callback)$), where a user thread is not blocked by its handling requests.

4.2 Balanced Request Allocation

To distribute user requests evenly among workers for efficient exploitation of inter-instance parallelism, p^{2}KVS adopts a simple and effective modular-based hash function to evenly divide the key-space, i.e., $W_{key} = Hash(key)\%N$, where W_{key} is the corresponding worker ID and N is the total number of workers that is predefined according to the actual measurement of hardware parallelism. We set 8 as the default total number of workers according to our hardware performance. This hash-based partition policy has three advantages: load balancing, minimal overhead, and no read magnification because there are no overlapping keys among partitions. Extending N or adjusting hash function may lead to a reconstruction of the entire set of KVS instances. More approaches of sophisticated hashing and runtime scaling (e.g., consistent hash [31]) will be further considered as a topic of our future study. Our experimental results show that even under highly skewed workloads with Zipfian distributions, the hash function can still make the hot requests evenly distributed across partitions. In the likelihood of imbalance among instances, e.g., most hot requests are occasionally hashed to a certain worker, p^{2}KVS is degraded to RocksDB with a single instance. Besides, p^{2}KVS can be configured with appropriate partition strategies to well match the access patterns of workloads, such as using multiple independent hash functions [39] or dynamic key-ranges [27].

Note that this key-range partitioning is equivalent to expanding the capacity of each level in the global LSM-tree with multiple mutually-exclusive sub-key-ranges. Therefore, the partitioning can reduce compaction-induced write amplification to some extent, because multiple instances increase the width of LSM-tree while reducing its depth [52].

4.3 Opportunistic Request Batching

As shown in Section 3.4, request batching is an effective method to reduce IO and CPU overhead for small-sized writes. In addition, some KVSs, such as RocksDB, have good parallel optimizations of read-type requests. These features help improve the overall performance of each worker.

To effectively leverage these intra-parallelism potentials, p^{2}KVS introduces a queue-based request batching scheduling technique called opportunistic batching mechanism (OBM), as shown in Figure 10. When a worker is handling requests, user threads add requests into the request queue. When the worker finishes processing a request, it checks the request queue. If there are two or more consecutive incoming requests of the same request-type (e.g., read-type GET or write-type PUT, UPDATE and DELETE), they are merged into a batched request that is then handled as a whole, as shown in Algorithm 1. For write-type requests, the worker processes them as a WriteBatch. Compared with IO-level batching such as RocksDB group logging, which merges the logging IOs of multiple user threads, this request-level batching is more beneficial for reducing the thread synchronization overheads. For read-type requests, the worker queries them concurrently on KVS. RocksDB provides a `multiget` interface which is a well-optimized operation to handle concurrent key search internally, and we use this interface in our implementation to handle read-type batched requests. Note that the worker does not proactively wait to capture incoming requests. Therefore, this batching is opportunistic.

The OBM can improve the processing efficiency under heavy concurrent workloads by eliminating the overhead of synchronization and waiting. To prevent the tail-latency problems due to extremely large batched-request, we set a predefined upper bound for the number of requests per batch (32 by default). Although different types of requests can also be processed in parallel within an instance, the OBM merges only same-type requests consecutively, to avoid consistency problems caused by out-of-order read-write requests when using the asynchronous interface. When the queues are often empty under light workloads, the approach simply degrades to the KVS without batching.

In summary, p^{2}KVS uses OBM to opportunistically aggregate multiple small requests into one larger request, which not only reduces the software and logging-IO overhead of the write process but also takes advantage of the parallel reading optimizations. Different from the IO-level batching mechanism in RocksDB or other KVSs [12, 35], p^{2}KVS avoids introducing additional synchronization overhead of merging writes in the underlying IO layer.

4.4 Range Query

Like other KVSs using hash indexes, it is a challenge for p^{2}KVS to implement range query operations (i.e., RANGE

Algorithm 1 Opportunistic batching.

Input: RQ, the request queue; M, the maximum batch size; $RList$, an empty request list;
Output: the return value of RocksDB instance.

```
function OPPORTUNISTIC_BATCHING(RQ, M, RList)
    R ← RQ.pop()
    if R.type = SCAN then
        return db.scan(R.begin, R.scansize)
    end if
    RList.add(R)
    batch_type ← R.type
    while (not RQ.empty()) and (RQ.first.type = batch_type) and (RList.size < M) do
        R ← RQ.pop()
        RList.add(R)
    end while
    if batch_type = WRITE then
        build a WriteBatch with all the requests in RList
        return write(WriteBatch)
    else if batch_type = READ then
        build a key list MergeKeys with all the request keys in RList
        return multiget(MergeKeys)
    end if
end function
```

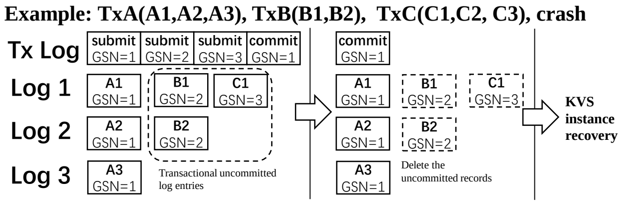


Figure 11. An example of transaction crash recovery.

and SCAN), because adjacent keys could be physically distributed to different instances. The key-space partitioning means that a range query must be forked into the corresponding workers covering the specified key range. Fortunately, each sharded instance using its own LSM-tree structure keeps its internal keys sorted, benefitting range queries.

There is a semantic difference between RANGE and SCAN, resulting in some differences in their implementations in p^{2}KVS. The RANGE operation specifies a beginning key and an end key and reads out all existing KV pairs between them. Differently, the SCAN operation specifies a begin key and the number of its subsequent KV pairs to read (i.e., scan-size). When the underlying IO bandwidth is sufficient, a RANGE request can be divided into multiple sub-RANGE operations executed by multiple instances of p^{2}KVS in parallel without extra cost. For the SCAN operation, the distribution of requested keys across the instances is unknown initially so that the number of target keys within each KVS instance is not determined *a priori*. A conservative way is to construct a global iterator based on the iterator of each KVS instance to serially traverse the keys in the entire key space like RocksDB `MergeIterator`. p^{2}KVS also provides an alternative parallelizing approach that first performs the SCAN operation with the same scan-size on all instances and then filters out the requested KVs from all return values. This approach causes extra reads, potentially impacting the performance. However, the simplicity and ease of implementation and the high bandwidth and parallelism offered by the underlying hardware may reasonably justify its use.

4.5 Crash Consistency

p^{2}KVS guarantees the same level of crash consistency as the underlying KVS instance, each of which can be recovered by replaying its own log file after a crash or failure. p^{2}KVS focuses on scaling the performance of KVS in its current design and can adopt existing transactional optimizations [36, 47] in the future.

Most LSM-tree based KVSs support basic transactions based on WriteBatch, where updates in the same transaction are committed by one WriteBatch. When a transaction covering multiple instances is executed, the transaction is split into multiple WriteBatches running on the instances in parallel, causing consistency problems if only a part of them are committed before crashing. To solve this problem, p^{2}KVS introduces a strictly increasing Global Sequence Number (GSN) for each write request to indicate its unique global order. GSN can be written to the KVS log files as a prefix of the original log sequence number. The WriteBatches split from the same transaction have the same GSN number and the OBM will not merge them with other requests.

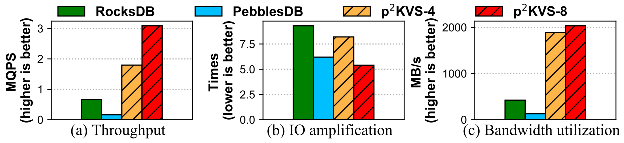
When an instance crashes, p^{2}KVS rolls back the logging requests in all instances according to the maximal GSN in the log of the crashed instance. To ensure recovery after a system-wide crash, p^{2}KVS persists the GSN of a transaction on the SSD when a transaction initializes or commits, thereby rolling back the whole transaction by canceling the corresponding WriteBatch on each KVS instance after a crash. For example, in Figure 11, before the crash, transaction A has returned and recorded the commit, transaction B has been processed by the KVSs but not committed, and transaction C has not been completed. When the system recovers, p^{2}KVS first deletes the log records of transactions B and C because the transaction log shows that the GSN of the last committed transaction is transaction A, and then performs the recovery process for all KVS instances. p^{2}KVS uses the snapshot feature of RocksDB to achieve the read committed transaction isolation level. Each worker creates a snapshot of the instance before the WriteBatch is processed, and other read requests will access the snapshot to avoid dirty reads. When the transaction commits, the snapshot will be deleted and the updates in the transaction will become visible.

5 Evaluation

We implement a prototype of p^{2}KVS upon RocksDB and evaluate it with both micro- and macro- benchmarks against state-of-the-art KVSs, including LSM-Tree based RocksDB

Table 1. Key characteristics of YCSB workloads. *RMW means a GET and an UPDATE that point to the same key.*

Workloads	Request ratio	Distribution	Count
LOAD	100% PUT	Uniform	670M
A	50% UPDATE 50% GET	Zipfian	120M
B	5% UPDATE 95% GET	Zipfian	120M
C	100% GET	Zipfian	120M
D	5% PUT 95% GET	Latest	120M
E	5% PUT 95% SCAN	Uniform	20M
F	50% RMW 50% GET	Zipfian	120M



(a) Throughput (b) IO amplification (c) Bandwidth utilization

Figure 12. Write performance. *Throughput, IO amplification and bandwidth utilization under random writes.*

and PebblesDB, and B-tree based KVell. PebblesDB [45] is a typical write optimized solution by reducing write amplification of compactions. KVell [35] exploits thread-level parallelism by maintaining multiple B-tree indexes with non-competitive worker threads (see Section 5.5 for details).

We also implement a LevelDB version of p^{2}KVS by replacing the `multiget` with multi-threaded GET operations in OBM because LevelDB does not support `multiget`. We compared its performance to LevelDB at the end of Section 5.2.

5.1 Experimental Setup

Hardware and Configuration. We run all experiments on a server with two Intel Xeon E5-2696 v4 CPUs (2.20 GHz, 22-core), 64 GB of DDR4 DRAM and an Intel Optane 905p 480 GB NVMe SSD. The Optane SSD exhibits high and stable write and read bandwidths of 2.2 GB/s and 2.6 GB/s, respectively. We use two configurations of p^{2}KVS with 4 or 8 workers, labeled *p^{2}KVS-4* and *p^{2}KVS-8*, respectively.

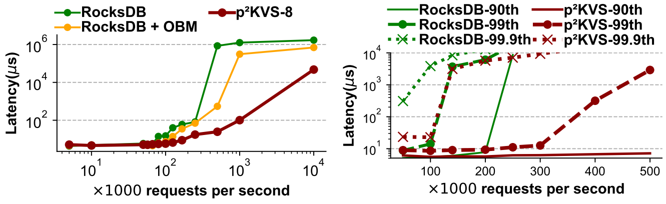
Workloads. We use micro-benchmarks to compare the peak processing capacity of p^{2}KVS and the baselines. We perform 100M random PUT operations using the `db_bench` tool with 16 user threads to evaluate concurrent write performance. The asynchronous interface of p^{2}KVS is enabled to show peak performance. We also perform 10M GET operations and 1M SCAN operations, respectively, to evaluate read performance. In macro-benchmarks, we use *YCSB* [15] to generate synthetic workloads, whose key characteristics are summarized in Table 1. We evaluate performance under both strong and weak concurrency in 2 sets of experiments with 8 and 32 user threads respectively. The size of KV pair is set to 128-Byte by default in both benchmarks.

5.2 Micro-benchmark

Write. We evaluate the application-level QPS and the storage-level IO bandwidth utilizations of RocksDB, PebblesDB, p^{2}KVS-4 and p^{2}KVS-8. As shown in Figure 12a, p^{2}KVS-4 and p^{2}KVS-8 outperform RocksDB in throughput by 2.7× and 4.6×, respectively.

Table 2. Memory and CPU usages under 100M random writes. *CPU usages are normalized to that of single-core.*

	Avg. Mem.	Max. Mem.	Avg. CPU	Max. CPU
RocksDB	0.14 GB	0.21 GB	1694%	1881%
PebblesDB	0.74 GB	1.93 GB	321%	441%
p^{2}KVS-4	0.58 GB	0.91 GB	762%	982%
p^{2}KVS-8	0.94 GB	1.20 GB	1239%	1417%

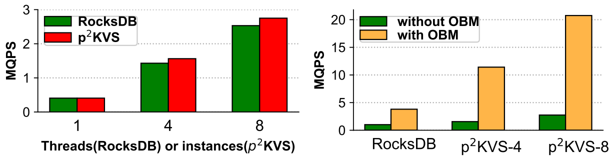


(a) Average latency **(b)** Tail latency

Figure 13. Write latencies as a function of different request intensities. *"RocksDB + OBM" in (a) denotes a single RocksDB instance with the opportunistic batching mechanism.*

To further analyze IO efficiency for applications, we also measure the IO amplification under the random write workload. Figure 12b shows that p^{2}KVS-8 has the lowest IO amplification due to the increased capacity at each level of its LSM-tree where a wider but shallower LSM-tree is formed collectively across all instances. Although PebblesDB optimizes compactions and achieves lower IO amplification than RocksDB and p^{2}KVS-4, it incurs higher IO amplification than p^{2}KVS-8 because it is developed based on LevelDB and not optimized for concurrent writes. To better understand the cause of performance enhancement achieved by p^{2}KVS, we examine the corresponding IO behaviors on SSD. As shown in Figure 12c, the SSD bandwidth is almost fully utilized with p^{2}KVS while the bandwidth usage of RocksDB and PebblesDB is less than 20%. This is because p^{2}KVS triggers compactions more frequently by eliminating the foreground bottlenecks. Clearly, p^{2}KVS' performance superiority comes from its highly effective capacity utilization of the underlying hardware.

We show in Table 2 the usages of memory and CPUs when processing 100M random writes on p^{2}KVS and other KVSs. The average memory usages are less than 1.5 GB across all KVSs. The CPU consumptions of p^{2}KVS-4 and p^{2}KVS-8 are over 7× and 12× that of a single core. These higher CPU usages of p^{2}KVS come from the 4 or 8 worker threads and additional background threads. The user threads sleep after submitting the request and only consume very little CPU resources. Each user thread in RocksDB overloads almost a whole CPU core, resulting in a huge CPU footprint at 16 threads. However, its throughput is low due to frequent thread synchronization and lock overhead. Therefore, the background compaction threads consume a small amount of CPU resources. Because LevelDB-based PebblesDB is not optimized for concurrent writes, most concurrent user threads are in a waiting state and only take up a small fraction of the total CPU resources.



(a) Without OBM

(b) Effect of OBM

Figure 14. Throughput of point query.

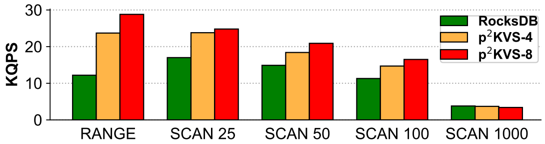


Figure 15. Throughputs of RANGE and SCAN operations with different scan sizes.

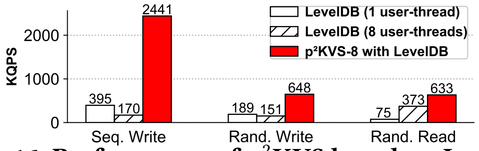


Figure 16. Performance of p^{2}KVS based on LevelDB.

Next, we assess the request latency as a function of load intensity. We conduct 1M random writes on RocksDB and p^{2}KVS with different request intensities. Figure 13a shows that the average latencies of p^{2}KVS and RocksDB are very close to each other under light workloads. However, p^{2}KVS can support much higher intensity at the same latency than RocksDB, because of its higher processing capacity. We further observe the tail latency, an important measure of the quality of user experience in KVS. As shown in Figure 13b, RocksDB suffers from drastic latency spikes when the request intensity exceeds 100 KQPS, while p^{2}KVS can guarantee 99th-percentile latency below 1 ms with less than 400 KQPS intensity.

Point query. Figure 14 shows the impact of multiple workers and OBM on point queries. We launch 10M GET requests to RocksDB and p^{2}KVS. As shown in Figure 14a, without OBM, p^{2}KVS has almost the same performance as RocksDB. By leveraging the read optimizations of RocksDB with OBM, p^{2}KVS achieves almost linear scalability as shown in Figure 14b. The QPS of p^{2}KVS-8 with OBM enabled outperforms the disabled case by up to 7.5×, and RocksDB by up to 5.4×.

Range Query. We load 100M 128-Byte KV pairs to warm up the system, and then perform 1M RANGE or SCAN operations with different scan sizes, using a single user thread. As shown in Figure 15, p^{2}KVS outperforms RocksDB by up to 2.9× in RANGE query. p^{2}KVS improves QPS by 1.5× during small-range SCAN because there is sufficient IO bandwidth to compensate for read amplification. When the scan-size is larger than 1000, p^{2}KVS with high read amplification saturates the SSD IO capacity, exhibiting the same performance as RocksDB. This demonstrates that p^{2}KVS further extends the benefit of parallelism based on the read optimizations of RocksDB.

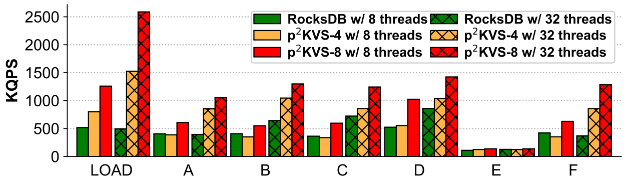


Figure 17. Throughputs under YCSB workloads.

p^{2}KVS based on leveldb. Figure 16 shows the throughputs of p^{2}KVS-8 based on LevelDB instances under micro-benchmarks. The results indicate that even though LevelDB does not offer intra-parallelism optimizations of RocksDB (e.g., pipelined write and multiget), p^{2}KVS can still improve LevelDB's random read and write performance by up to 8.4× and 3.4×, respectively.

5.3 Macro-benchmark

We evaluate the effectiveness of p^{2}KVS relative to RocksDB under YCSB workloads with 8 or 32 threads, as shown in Figure 17. Results of PebblesDB are excluded because it fills up all memory and crashes when writing hundreds of millions of KV pairs.

Under write-intensive workloads (LOAD), with more user threads, p^{2}KVS exhibits higher speedup. For example, p^{2}KVS-8 outperforms RocksDB by 2.4× and 5.2× with 8 and 32 user threads, respectively. This is because p^{2}KVS not only incorporates RocksDB's request-level batching optimization by OBM but also improves parallelism efficiency through multiple non-competitive workers, especially under high concurrent workloads (e.g., 32 threads). The performance improvement of p^{2}KVS-8 is more obvious than that of p^{2}KVS-4, indicating that the number of workers should match the hardware parallelism to maximize performance.

Under read-intensive workloads (B, C, D), p^{2}KVS improves over RocksDB by about 1~2× in QPS. This read performance improvement comes from not only using OBM to leverage RocksDB's original read parallelism, but also from the additional benefits of the hash partitioned indexes and the parallel workers. Under workload E (i.e., 95% SCAN and 5% PUT), the performance of p^{2}KVS is similar to that of RocksDB because p^{2}KVS'gain in exploiting IO parallelism is offset by the resulting read amplification.

Under mixed workloads (A, F), p^{2}KVS outperforms RocksDB by 1.5~3.5× mainly because of the optimization for the concurrent write process.

5.4 Sensitivity Study

We perform sensitivity studies to understand the impact of different design parameters and choices of p^{2}KVS on the overall performance. We still use YCSB as workloads and take RocksDB with a single user thread as a baseline.

Number of workers and OBM. We vary the number of workers with enabled and disabled OBM. The result is shown in Figure 18. Under the write-intensive workload LOAD, the

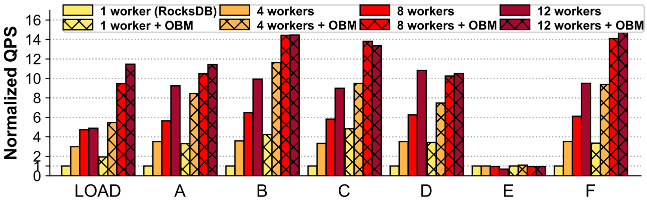


Figure 18. p^{2}KVS sensitivity to the number of workers and OBM. *The single-worker configuration is equivalent to RocksDB. All throughputs are normalized to RocksDB. Solid-colored and patterned bars are used to indicate OBM being disabled and enabled, respectively.*

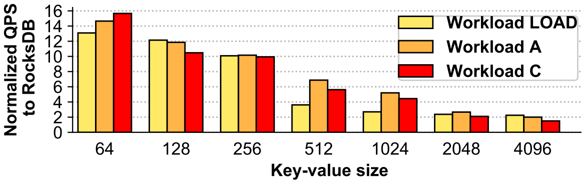


Figure 19. p^{2}KVS sensitivity to key-value size.

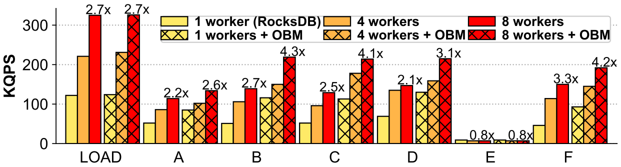


Figure 20. p^{2}KVS performance at key-value size of 1KB.

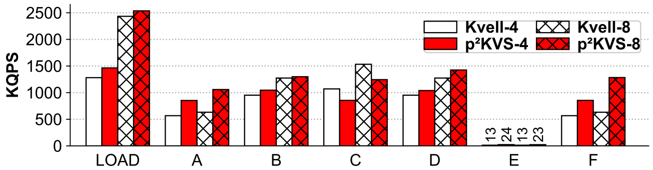
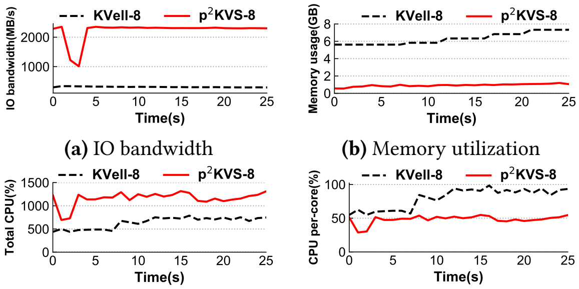


Figure 21. Overall performance of KVell and p^{2}KVS.



(a) IO bandwidth **(b)** Memory utilization

(c) Total CPU utilization **(d)** Per-core CPU utilization

Figure 22. Hardware utilizations of p^{2}KVS and KVell under random write workloads.

inter-instance parallelism helps boost performance by 3× and 5× with 4 and 8 instances, respectively. OBM further accelerates the write speed by exploiting request batching, increasing QPS by up to 2×. The results suggest that simply increasing the number of instances cannot significantly improve the performance in cases of small-sized KVs. However, when applying OBM, the write throughput scales well.

Under the read-intensive workload C, the inter-instance parallelizing improves performance by up to 3.3× and 5.8×, with 4 and 8 workers, respectively. OBM improves read performance by 5× even with a single instance, but only achieves 2× QPS enhancement with 8 workers because parallelism at this level has nearly exhausted SSD capacity, leaving insufficient bandwidth to be further leveraged by OBM.

Under hybrid workloads A and B, without OBM the overall performance increases by up to 3.5× and 6.5× under 4 and 8 instances, respectively. OBM increases QPS by 2.2×~4.2× under workload B while gaining less benefit under workload A, increasing the throughput by 1.8×~3.2×. This is because workload B has more mixed read and write requests, limiting the batching size of OBM that only batches adjacent requests of the same type.

Key-value Size. Next, we observe the impact of different KV sizes. We test the performance as a function of KV-sizes in three typical workloads (LOAD, A, and C), as shown in Figure 19. The result demonstrates that the small-KV cases benefit more from OBM than the large-KV cases. Figure 20 shows the performance of p^{2}KVS at the KV size of 1KB, which exhibits a lower speedup than the case at 128-Byte KV. OBM is less effective under write-intensive workloads with large-sized KV because the benefit of merging large logging IOs is small. However, OBM remains effective for read-intensive workloads.

5.5 p^{2}KVS vs. KVell

KVell uses multiple workers to maintain multiple independent B-tree indexes that can be accessed in parallel. It uses in-place updates for all write requests to avoid write amplification, and maintains large indexes and page cache in memory to speed up queries. We compare KVell with 4 or 8 worker-threads to p^{2}KVS-4 and p^{2}KVS-8 by using macro-benchmarks, as shown in Figure 21. We configure the page cache size of KVell to 4 GB, which consumes an acceptable amount of memory and is much larger than 8 MB block cache of each RocksDB instance. Even with this configuration, the maximum memory consumption of KVell is 22 GB due to the large in-memory indexes, in contrast to the 3 GB memory consumption of p^{2}KVS. Under write-intensive workloads (LOAD, A, and F) the performance of p^{2}KVS is higher than KVell. p^{2}KVS's point query performance is similar to KVell (under workloads B and D) and its SCAN performance is higher than KVell (under workload E). Under workload C, KVell achieves higher throughputs than p^{2}KVS due to its large page cache and all-in-memory indexes.

We also record and compare the utilizations of IO bandwidth, memory, and CPU of p^{2}KVS-8 and KVell-8 under a continuous 100M random write workload. The throughputs of KVell-8 and p^{2}KVS-8 are 2.5 MQPS and 3.0 MQPS respectively. As shown in Figure 22a, although KVell uses all-random IO without write amplification and high IO bandwidth utilization under 1KB KV-size, it only consumes about 300 MB/s IO bandwidth under small-sized 128-byte KV writes.

In contrast, LSM-tree is more suitable for aggregating small IOs and enables p^{2}KVS to utilize full IO bandwidth. Figure 22b shows that even after subtracting the footprint of the page cache, KVell still uses 2× more memory than p^{2}KVS because it stores all the indexes in memory, while LSM-tree sorts the data on disk to reduce the index size. Meanwhile, each thread of KVell maintains a large index, resulting in an average CPU utilization per core of more than 80%. However, each RocksDB instance under p^{2}KVS runs multiple threads in the foreground and background to perform logging and compactions separately. Therefore, although the total CPU utilization of p^{2}KVS is higher, each core consumes about 50% of CPU as shown in Figure 22c and 22d. This means that p^{2}KVS is more suitable for multicore hardware environments and does not rely on single-core performance.

6 Related Works

A large body of research has been conducted to improve KVS performance on SSDs.

Optimize WAL and MemTable Some existing works attempt to directly address the problem of low performance in WAL or MemTable steps. Faster [10] stores a part of the log in volatile DRAM and persists it asynchronously to increase WAL performance. Aether [28] takes sophisticated approaches such as Early Lock Release and Flush Pipelining to reduce concurrency-induced contention for small-sized log writes. Taurus [49] further optimizes these techniques and implements an efficient parallel logging scheme by tracking and encoding transactions with the log sequence number. SpanDB [12] provides asynchronous group logging and request processing with polling-based IO via SPDK by using asynchronous request interfaces.

FloDB [5], Accordion [6], WipDB [56], and CruiseDB [38] improve MemTable's write performance by modifying the data structure of the memory component. FloDB and Accordion add buffers on the top of skiplist-based MemTable. WipDB replaces skiplist with multiple large hash tables and compacts KV-pairs in memory instead of in SSD. CruiseDB dynamically adjusts the size of MemTable based on workloads and SLA to reduce write stalls. p^{2}KVS is compatible with these works and can absorb their ideas while using efficient parallel scheduling to avoid the contentions by WAL and on in-memory structures.

Optimize LSM-tree Compacting. In order to efficiently utilize the bandwidth of SSDs, many solutions reduce the write amplification by modifying the LSM-tree structure. PebblesDB [45] designs a fragmented LSM-tree structure, which allows overlapping key-ranges on tree levels and reduces most of the compaction overhead. LSM-trie [48], SifrDB [42], Dostoevsky [18] and SlimDB [46] also design some variants of LSM-tree to mitigate write amplification by tolerating overlapping key-ranges. ForestDB [1], WiscKey [40], LWC-Tree [51], HashKV [9], UniKV [55] and diffKV [37] apply KV separation mechanism, which stores KV pairs in multiple log files and records key-pointer pairs in LSM-tree levels, thereby reducing write amplification for large-sized KV pairs.

While sharing a common goal to optimize the IO efficiency of LSM-tree-based KVSs, p^{2}KVS, as a user-space scheduler, is orthogonal and complementary to these solutions and highly portable for implementation on top of existing KVSs.

Design Non-LSM-tree. Some studies have designed new structures for high-performance SSDs in lieu of LSM-tree. KVell [35] maintains large B-tree-based indexes and page cache in memory to ensure GET and SCAN performance on modern fast SSDs. uDepot [32] stores data in unordered segments mapped by a hash table and fully utilizes SSDs by leveraging asynchronous user-space IO with a task runtime system. Tucana [44] uses B$^\epsilon$-tree to reduce the overhead and IO amplification of compactions. SplinterDB [14] designs STB$^\epsilon$-tree based on B$^\epsilon$-tree, which is optimized for SSDs with high hardware parallelism. Although these new indexing structures take full advantage of modern SSDs, p^{2}KVS takes an orthogonal approach that views KVS and SSD as black boxes and thus inherits time-tested, desirable features of the widely-used and well-optimized LSM-tree based KVSs (e.g., RocksDB) and SSDs, providing high portability.

Sharding KVS. Distributed databases partition table-space into multiple tablets and store them in different KVS instances among nodes [2, 11, 27, 57]. Recently, LSM-tree based OLTP storage engine running upon high-performance hardware employs multiple LSM-tree instances, each of which is used to store a table, or subtable, or an index[19, 26, 50]. Specific interface semantics (e.g., column) or dynamic scheduling policies are used to determine the instances where key-value pairs are placed. However, p^{2}KVS uses the key-space sharding to evenly assign KVs to multiple workers without database semantics to expedite the global KVS.

7 Conclusion

In the production-level key-value store environments, system administrators expect to obtain consistent performance improvement by simply replacing slow HDDs with fast SSDs. The outcome has often been disappointing, especially for pervasive small-sized KV workloads. We reveal that the foreground operations in the KVS write process (logging and indexing) can become a serious performance bottleneck under single-threaded and concurrent write workloads. We present a portable parallelizing engine, p^{2}KVS, upon multiple instances of LSM-tree based KVS to effectively and efficiently perform KV operations. p^{2}KVS is designed to exploit inherent parallelism among and within these instances to fully utilize the processing capacities offered by modern CPU, memory, and storage hardware. Compared to the state-of-the-art vanilla RocksDB, p^{2}KVS improves write performance and read performance by up to 4.6× and 5.4× respectively.

References

[1] Jung-Sang Ahn, Chiyoung Seo, Ravi Mayuram, Rahim Yaseen, Jin-Soo Kim, and Seungryoul Maeng. 2016. ForestDB: A Fast Key-Value Storage System for Variable-Length String Keys. *IEEE Trans. Comput.* 65, 3 (March 2016), 902–915.

[2] Apache. 2021. HBase. https://hbase.apache.org/.

[3] Oana Balmau, Diego Didona, Rachid Guerraoui, Willy Zwaenepoel, Huapeng Yuan, Aashray Arora, Karan Gupta, and Pavan Konka. 2017. TRIAD: Creating Synergies Between Memory, Disk and Log in Log Structured Key-Value Stores. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. USENIX Association, Santa Clara, CA, 363–375. https://www.usenix.org/conference/atc17/technical-sessions/presentation/balmau

[4] Oana Balmau, Florin Dinu, Willy Zwaenepoel, Karan Gupta, Ravishankar Chandhiramoorthi, and Diego Didona. 2019. SILK: Preventing Latency Spikes in Log-Structured Merge Key-Value Stores. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, Renton, WA, 753–766. https://www.usenix.org/conference/atc19/presentation/balmau

[5] Oana Balmau, Rachid Guerraoui, Vasileios Trigonakis, and Igor Zablotchi. 2017. FloDB: Unlocking Memory in Persistent Key-Value Stores. In *Proceedings of the Twelfth European Conference on Computer Systems, EuroSys 2017, Belgrade, Serbia, April 23-26, 2017*, Gustavo Alonso, Ricardo Bianchini, and Marko Vukolic (Eds.). ACM, 80–94. https://doi.org/10.1145/3064176.3064193

[6] Edward Bortnikov, Anastasia Braginsky, Eshcar Hillel, Idit Keidar, and Gali Sheffi. 2018. Accordion: Better Memory Organization for LSM Key-Value Stores. *Proc. VLDB Endow.* 11, 12 (2018), 1863–1875. https://doi.org/10.14778/3229863.3229873

[7] Zhichao Cao, Siying Dong, Sagar Vemuri, and David H.C. Du. 2020. Characterizing, Modeling, and Benchmarking RocksDB Key-Value Workloads at Facebook. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*. USENIX Association, Santa Clara, CA, 209–223. https://www.usenix.org/conference/fast20/presentation/cao-zhichao

[8] Yunpeng Chai, Yanfeng Chai, Xin Wang, Haocheng Wei, Ning Bao, and Yushi Liang. 2019. LDC: A Lower-Level Driven Compaction Method to Optimize SSD-Oriented Key-Value Stores. In *35th IEEE International Conference on Data Engineering, ICDE 2019, Macao, China, April 8-11, 2019*. IEEE, 722–733. https://doi.org/10.1109/ICDE.2019.00070

[9] Helen H. W. Chan, Yongkun Li, Patrick P. C. Lee, and Yinlong Xu. 2018. HashKV: Enabling Efficient Updates in KV Storage via Hashing. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, Boston, MA, 1007–1019. https://www.usenix.org/conference/atc18/presentation/chan

[10] Badrish Chandramouli, Guna Prasaad, Donald Kossmann, Justin J. Levandoski, James Hunter, and Mike Barnett. 2018. FASTER: A Concurrent Key-Value Store with In-Place Updates. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein (Eds.). ACM, 275–290. https://doi.org/10.1145/3183713.3196898

[11] 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* 26, 2 (June 2008), 1–26.

[12] Hao Chen, Chaoyi Ruan, Cheng Li, Xiaosong Ma, and Yinlong Xu. 2021. SpanDB: A Fast, Cost-Effective LSM-tree Based KV Store on Hybrid Storage. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*. USENIX Association, 17–32. https://www.usenix.org/conference/fast21/presentation/chen-hao

[13] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. 2015. MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems. *CoRR* abs/1512.01274 (2015). arXiv:1512.01274 http://arxiv.org/abs/1512.01274

[14] Alexander Conway, Abhishek Gupta, Vijay Chidambaram, Martin Farach-Colton, Richard Spillane, Amy Tai, and Rob Johnson. 2020. SplinterDB: Closing the Bandwidth Gap for NVMe Key-Value Stores. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, 49–63. https://www.usenix.org/conference/atc20/presentation/conway

[15] 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*. ACM, 143–154.

[16] Tyler Crain, Vincent Gramoli, and Michel Raynal. 2013. No Hot Spot Non-blocking Skip List. In *IEEE 33rd International Conference on Distributed Computing Systems, ICDCS 2013, 8-11 July, 2013, Philadelphia, Pennsylvania, USA*. IEEE Computer Society, 196–205. https://doi.org/10.1109/ICDCS.2013.42

[17] Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. 2015. Asynchronized Concurrency: The Secret to Scaling Concurrent Search Data Structures. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '15, Istanbul, Turkey, March 14-18, 2015*, Özcan Özturk, Kemal Ebcioglu, and Sandhya Dwarkadas (Eds.). ACM, 631–644. https://doi.org/10.1145/2694344.2694359

[18] Niv Dayan and Stratos Idreos. 2018. Dostoevsky: Better Space-Time Trade-Offs for LSM-Tree Based Key-Value Stores via Adaptive Removal of Superfluous Merging. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein (Eds.). ACM, 505–520. https://doi.org/10.1145/3183713.3196927

[19] Siying Dong, Andrew Kryczka, Yanqin Jin, and Michael Stumm. 2021. Evolution of Development Priorities in Key-value Stores Serving Large-scale Applications: The RocksDB Experience. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*. USENIX Association, 33–49. https://www.usenix.org/conference/fast21/presentation/dong

[20] Facebook. 2015. Column Families of RocksDB. https://github.com/facebook/rocksdb/wiki/Column-Families.

[21] Facebook. 2021. RocksDB. https://rocksdb.org/.

[22] Facebook. 2021. RocksDB tuning guide. https://github.com/facebook/rocksdb/wiki/RocksDB-Tuning-Guide.

[23] Google. 2020. LevelDB. https://github.com/google/leveldb.

[24] Shukai Han, Dejun Jiang, and Jin Xiong. 2020. LightKV: A Cross Media Key Value Store with Persistent Memory to Cut Long Tail Latency. In *36th Symposium on Mass Storage Systems and Technologies, MSST 2020, Santa Clara, CA, USA, October 29, 2020*. 1–13.

[25] Shukai Han, Dejun Jiang, and Jin Xiong. 2020. SplitKV: Splitting IO Paths for Different Sized Key-Value Items with Advanced Storage Devices. In *12th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 20)*. USENIX Association. https://www.usenix.org/conference/hotstorage20/presentation/han

[26] Gui Huang, Xuntao Cheng, Jianying Wang, Yujie Wang, Dengcheng He, Tieying Zhang, Feifei Li, Sheng Wang, Wei Cao, and Qiang Li. 2019. X-Engine: An Optimized Storage Engine for Large-scale E-commerce Transaction Processing. In *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, Peter A. Boncz, Stefan Manegold, Anastasia Ailamaki, Amol Deshpande, and Tim Kraska (Eds.). ACM, 651–665. https://doi.org/10.1145/3299869.3314041

[27] Haoyu Huang and Shahram Ghandeharizadeh. 2021. Nova-LSM: A Distributed, Component-based LSM-tree Key-value Store. In *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, Guoliang Li, Zhanhuai Li, Stratos Idreos, and

Divesh Srivastava (Eds.). ACM, 749–763. https://doi.org/10.1145/3448016.3457297

[28] Ryan Johnson, Ippokratis Pandis, Radu Stoica, Manos Athanassoulis, and Anastasia Ailamaki. 2010. Aether: A Scalable Approach to Logging. *Proc. VLDB Endow.* 3, 1 (2010), 681–692. https://doi.org/10.14778/1920841.1920928

[29] Olzhas Kaiyrakhmet, Songyi Lee, Beomseok Nam, Sam H. Noh, and Young ri Choi. 2019. SLM-DB: Single-Level Key-Value Store with Persistent Memory. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*. USENIX Association, Boston, MA, 191–205. https://www.usenix.org/conference/fast19/presentation/kaiyrakhmet

[30] Sudarsun Kannan, Nitish Bhat, Ada Gavrilovska, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. 2018. Redesigning LSMs for Nonvolatile Memory with NoveLSM. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, Boston, MA, 993–1005. https://www.usenix.org/conference/atc18/presentation/kannan

[31] David R. Karger, Eric Lehman, Frank Thomson Leighton, Rina Panigrahy, Matthew S. Levine, and Daniel Lewin. 1997. Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web. In *Proceedings of the Twenty-Ninth Annual ACM Symposium on the Theory of Computing, El Paso, Texas, USA, May 4-6, 1997*, Frank Thomson Leighton and Peter W. Shor (Eds.). ACM, 654–663. https://doi.org/10.1145/258533.258660

[32] Kornilios Kourtis, Nikolas Ioannou, and Ioannis Koltsidas. 2019. Reaping the performance of fast NVM storage with uDepot. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*. USENIX Association, Boston, MA, 1–15. https://www.usenix.org/conference/fast19/presentation/kourtis

[33] Chunbo Lai, Song Jiang, Liqiong Yang, Shiding Lin, Guangyu Sun, Zhenyu Hou, Can Cui, and Jason Cong. 2015. Atlas: Baidu's key-value storage system for cloud data. In *IEEE 31st Symposium on Mass Storage Systems and Technologies, MSST 2015, Santa Clara, CA, USA, May 30 - June 5, 2015*. IEEE Computer Society, 1–14. https://doi.org/10.1109/MSST.2015.7208288

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

[35] Baptiste Lepers, Oana Balmau, Karan Gupta, and Willy Zwaenepoel. 2019. KVell: the design and implementation of a fast persistent key-value store. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP 2019, Huntsville, ON, Canada, October 27-30, 2019*, Tim Brecht and Carey Williamson (Eds.). ACM, 447–461. https://doi.org/10.1145/3341301.3359628

[36] Baptiste Lepers, Oana Balmau, Karan Gupta, and Willy Zwaenepoel. 2020. Kvell+: Snapshot Isolation without Snapshots. In *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020*. USENIX Association, 425–441. https://www.usenix.org/conference/osdi20/presentation/lepers

[37] Yongkun Li, Zhen Liu, Patrick P. C. Lee, Jiayu Wu, Yinlong Xu, Yi Wu, Liu Tang, Qi Liu, and Qiu Cui. 2021. Differentiated Key-Value Storage Management for Balanced I/O Performance. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. USENIX Association, 673–687. https://www.usenix.org/conference/atc21/presentation/li-yongkun

[38] Junkai Liang and Yunpeng Chai. 2021. CruiseDB: An LSM-Tree Key-Value Store with Both Better Tail Throughput and Tail Latency. In *37th IEEE International Conference on Data Engineering, ICDE 2021, Chania, Greece, April 19-22, 2021*. IEEE, 1032–1043. https://doi.org/10.1109/ICDE51399.2021.00094

[39] Zaoxing Liu, Zhihao Bai, Zhenming Liu, Xiaozhou Li, Changhoon Kim, Vladimir Braverman, Xin Jin, and Ion Stoica. 2019. DistCache: Provable Load Balancing for Large-Scale Storage Systems with Distributed Caching. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*. USENIX Association, Boston, MA, 143–157. https://www.usenix.org/conference/fast19/presentation/liu

[40] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2016. WiscKey: Separating Keys from Values in SSD-conscious Storage. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*. USENIX Association, Santa Clara, CA, 133–148. https://www.usenix.org/conference/fast16/technical-sessions/presentation/lu

[41] Yoshinori Matsunobu, Siying Dong, and Herman Lee. 2020. MyRocks: LSM-Tree Database Storage Engine Serving Facebook's Social Graph. *Proc. VLDB Endow.* 13, 12 (2020), 3217–3230. http://www.vldb.org/pvldb/vol13/p3217-matsunobu.pdf

[42] Fei Mei, Qiang Cao, Hong Jiang, and Jingjun Li. 2018. SifrDB: A Unified Solution for Write-Optimized Key-Value Stores in Large Datacenter. In *Proceedings of the ACM Symposium on Cloud Computing, SoCC 2018, Carlsbad, CA, USA, October 11-13, 2018*. ACM, 477–489. https://doi.org/10.1145/3267809.3267829

[43] Patrick E. O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth J. O'Neil. 1996. The Log-Structured Merge-Tree (LSM-Tree). *Acta Informatica* 33, 4 (1996), 351–385. https://doi.org/10.1007/s002360050048

[44] Anastasios Papagiannis, Giorgos Saloustros, Pilar González-Férez, and Angelos Bilas. 2016. Tucana: Design and Implementation of a Fast and Efficient Scale-up Key-value Store. In *2016 USENIX Annual Technical Conference, USENIX ATC 2016, Denver, CO, USA, June 22-24, 2016*, Ajay Gulati and Hakim Weatherspoon (Eds.). USENIX Association, 537–550. https://www.usenix.org/conference/atc16/technical-sessions/presentation/papagiannis

[45] Pandian Raju, Rohan Kadekodi, Vijay Chidambaram, and Ittai Abraham. 2017. PebblesDB: Building Key-Value Stores using Fragmented Log-Structured Merge Trees. In *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017*. ACM, 497–514. https://doi.org/10.1145/3132747.3132765

[46] Kai Ren, Qing Zheng, Joy Arulraj, and Garth Gibson. 2017. SlimDB: A Space-Efficient Key-Value Storage Engine For Semi-Sorted Data. *Proc. VLDB Endow.* 10, 13 (2017), 2037–2048. https://doi.org/10.14778/3151106.3151108

[47] Cheng Tan, Changgeng Zhao, Shuai Mu, and Michael Walfish. 2020. Cobra: Making Transactional Key-Value Stores Verifiably Serializable. In *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020*. USENIX Association, 63–80. https://www.usenix.org/conference/osdi20/presentation/tan

[48] Xingbo Wu, Yuehai Xu, Zili Shao, and Song Jiang. 2015. LSM-trie: An LSM-tree-based Ultra-Large Key-Value Store for Small Data Items. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*. USENIX Association, Santa Clara, CA, 71–82. https://www.usenix.org/conference/atc15/technical-session/presentation/wu

[49] Yu Xia, Xiangyao Yu, Andrew Pavlo, and Srinivas Devadas. 2020. Taurus: Lightweight Parallel Logging for In-Memory Database Management Systems. *Proc. VLDB Endow.* 14, 2 (2020), 189–201. https://doi.org/10.14778/3425879.3425889

[50] Baoyue Yan, Xuntao Cheng, Bo Jiang, Shibin Chen, Canfang Shang, Jianying Wang, Kenry Huang, Xinjun Yang, Wei Cao, and Feifei Li. 2021. Revisiting the Design of LSM-tree Based OLTP Storage Engine with Persistent Memory. *Proc. VLDB Endow.* 14, 10 (2021), 1872–1885. http://www.vldb.org/pvldb/vol14/p1872-yan.pdf

[51] Ting Yao, Jiguang Wan, Ping Huang, Xubin He, Fei Wu, and Changsheng Xie. 2017. Building Efficient Key-Value Stores via a Lightweight Compaction Tree. *ACM Trans. Storage* 13, 4 (2017), 29:1–29:28. https://doi.org/10.1145/3139922

[52] Ting Yao, Yiwen Zhang, Jiguang Wan, Qiu Cui, Liu Tang, Hong Jiang, Changsheng Xie, and Xubin He. 2020. MatrixKV: Reducing Write Stalls and Write Amplification in LSM-tree Based KV Stores with Matrix Container in NVM. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, 17–31. https://www.usenix.org/conference/atc20/presentation/yao

[53] Yugabyte. 2021. DocDB. https://blog.yugabyte.com/enhancing-rocksdb-for-speed-scale/.

[54] Deli Zhang and Damian Dechev. 2016. A Lock-Free Priority Queue Design Based on Multi-Dimensional Linked Lists. *IEEE Trans. Parallel Distributed Syst.* 27, 3 (2016), 613–626. https://doi.org/10.1109/TPDS.2015.2419651

[55] Qiang Zhang, Yongkun Li, Patrick P. C. Lee, Yinlong Xu, Qiu Cui, and Liu Tang. 2020. UniKV: Toward High-Performance and Scalable KV Storage in Mixed Workloads via Unified Indexing. In *36th IEEE International Conference on Data Engineering, ICDE 2020, Dallas, TX, USA, April 20-24, 2020.* IEEE, 313–324. https://doi.org/10.1109/ICDE48307.2020.00034

[56] Xingsheng Zhao, Song Jiang, and Xingbo Wu. 2021. WipDB: A Write-in-place Key-value Store that Mimics Bucket Sort. In *37th IEEE International Conference on Data Engineering, ICDE 2021, Chania, Greece, April 19-22, 2021.* IEEE, 1404–1415. https://doi.org/10.1109/ICDE51399.2021.00125

[57] Tao Zhu, Zhuoyue Zhao, Feifei Li, Weining Qian, Aoying Zhou, Dong Xie, Ryan Stutsman, HaiNing Li, and Huiqi Hu. 2019. SolarDB: Toward a Shared-Everything Database on Distributed Log-Structured Storage. *ACM Trans. Storage* 15, 2 (2019), 11:1–11:26. https://doi.org/10.1145/3318158

A Artifact Appendix

A.1 Abstract

Our artifact consists of two parts. The first part is a script to analyze the bottlenecks of RocksDB and the second is a simple prototype of p^2KVS with some benchmarks.

A.2 Description & Requirements

A.2.1 How to access. *We store the artifact code in our GitHub repository https://github.com/luziyi23/p2KVS.*

A.2.2 Hardware dependencies. *The platform should be equipped with at least one multi-core CPU(preferably at least 16 cores) and a high-performance SSD(at least 2GB/s read/write bandwidth). Note that the choice of hardware will affect the test results.*

A.2.3 Software dependencies. *Any distribution of Linux. We recommend Linux versions above 5.4.*

A.2.4 Benchmarks. *We use* `db_bench` *and YCSB[15] to generate workloads for evaluation. They have already been integrated into the repository.*

A.3 Set-up

git clone. See README file in the code directory.

A.4 Evaluation workflow

A.4.1 Major Claims.

- *(C1): As the number of user threads increases, RocksDB's throughput increases only moderately (3× speedup at 32 threads as stated) because thread synchronization overheads cause the bottleneck. This is proven by the experiment (E1) described in Section 3.2 and Section 3.3 whose results are reported in Figure 5a and Figure 6.*
- *(C2): p^2KVS with 8 workers improves the write performance of RocksDB by up to 4.6×. This is proven by the experiments (E2) in Section 5.2 whose illustrated in Figure 12a.*

A.4.2 Experiments. *Experiment (E1): [Bottleneck Analysis] [10 human-minutes + 10 compute-minutes]: Evaluate the random write throughput of RocksDB for different user threads and analyze bottlenecks by latency breakdown.*

[Preparation] Build RocksDB and DB_Bench benchmarks. Install the Linux-perf and FlameGraph for latency breakdown. Mount the high-performance SSD in any directory.

[Execution] Change the `db_dir` *variable in* `scripts/perf.py` *to the SSD directory. This script will evaluate RocksDB by performing 10M writes at 1 to 32 user-threads respectively and save the latency breakdown results as flame graphs.*

[Results] All throughput and average latency results are in `test_result.txt`*, while the latency breakdown graph for each experiment are in* `perf_[#threads].svg`.

Experiment (E2): [Write Performance Evaluation] [10 human-minutes + 10 compute-minutes]: Evaluate the write performance of p^2KVS-8 with micro-benchmarks.

[Preparation] Build the test code with the following instruction.
`make write_test DATANUM=10000000 INSTNUM=8`

[Execution] Run the test with the following instruction.
`write_test [the SSD directory].`

[Results] The program will output results such as throughput and latency on the screen. Compare the throughput with the highest write throughput of RocksDB in (E1) to get the performance improvement of p2KVS relative to RocksDB.