# SAKER: A Software Accelerated Key-value Service via the NVMe Interface

Chen Zhong
University of Texas at Arlington
chen.zhong@mavs.uta.edu

Wenguang Wang
Broadcom Inc.
wenguang.wang@broadcom.com

Song Jiang
University of Texas at Arlington
song.jiang@uta.edu

## Abstract

The NVMe Key Value (NVMe-KV) Command Set has been standardized to enable access to an NVMe device with a key rather than a block address and make an NVMe device a KV service provider. This new interface opens an exciting opportunity of offloading extensive data management chores to an external KV device and streamlining the KV-based data processing at the host. However, the interface itself may become a major performance bottleneck with small KV access and make the technology hard to be deployed in diverse application scenarios. In this paper we proposed a software-based facility, named SAKER, at the host side to remove or alleviate the performance bottleneck at the interface. SAKER, which was prototyped in an NVMe-KV SSD emulator, demonstrates that it can effectively keep the NVMe-KV interface from becoming the performance bottleneck even with small KV requests in most workloads.

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

***Keywords:*** NVMe-KV, Key-value Store,

## 1 Introduction

Enterprise storage is a multi-trillion-dollar market with fierce competition among vendors. Having a standardized storage interface is the key to establishing and growing the market. For example, SCSI and NVMe are the standards for block interfaces, NFS and SMB are the standards for POSIX file interfaces. Storage vendors are willing to compete at a level higher up in the storage stack in order to offer more value and differentiation. Key-value (KV) stores, as a data management service, have become an indispensable storage infrastructure on which various applications can be developed. For example, many DBMSes, such as MySQL [11] and BigTable [2], use a KV store as their storage backend. However, without a set standard for KV store interface, storage vendors cannot provide KV service to upper layers and monetize it.

*SYSTOR '25, Virtual, Israel*
© 2025 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2119-9/2025/09
https://doi.org/10.1145/3757347.3759143

Furthermore, if a highly performant standard interface for KV service is introduced, developers can focus on business logic above the KV layer to reduce development time. In the meantime, storage vendors can compete by offering the standard KV service instead of the primitive block storage service. When storage vendors can optimize the full stack of KV service, better designs may emerge to provide more cost-effective and faster service on top of it. Recently, the NVMe Key Value (NVMe-KV) Command Set was standardized by the NVM Express Work Group, which specifies a common set of KV access commands including read, write, and delete keys [23]. Samsung's KV-SSD that maintains a KV store within the SSD with a KV access interface has become available. With the emergence of the standard and increasing support from vendors, it's expected that more NVMe KV-interfaced SSDs, storage appliances, and disaggregated storage services will appear. Because NVMe can be a virtual device running in virtual machines, or as the NVMe initiator connecting to a remote NVMe target via the NVMe-over-Fabric (NVMe-oF) protocol, any traditional disk array vendor can adopt either virtual NVMe KV interface or NVMe KV via NVMe-oF and compete in this emerging and bigger market.

However, even with its clear technological and economic advantages, the success of the NVMe KV storage hinges on whether it can provide performance competitive to that of the KV store on the host. To help reveal the performance gap between an NVMe-based KV SSD and a local on-host KV store, we experimented with KV writes of different sizes to an NVMe SSD emulator [9]. The emulator can be configured with different sets of performance specs. In particular, we configured three SSDs, each representing a different speed class (high, medium, and low). Their key performance specs are shown in Figure 2, which are taken from the PCIe 3.0 [16], 4.0 [17], and 5.0 [18] SSDs, respectively. In the service of write requests, the hash-table-based KV store writes its received KV items into an on-flash log once an in-memory buffer block is filled. Keys on the log are indexed by an in-memory hash table. Figure 1 shows the throughput of a million random write requests to the KV store running on the host server (*HostKV* for short) or one that resides in an SSD disk (*DiskKV* for short). The key size is 16 bytes. The value size varies from 16B to 4KB. With a small value (16B), HostKV achieves over 5.5× higher throughput than DiskKV. The performance gap reduces with larger value sizes. However, the faster an SSD is, the smaller the reduction is. For example, for the value

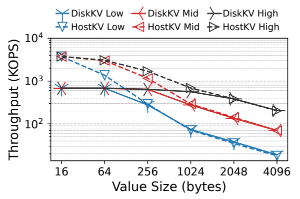

**Figure 1.** Low, Mid and High speed NVMe SSDs

| | Low | Mid | High |
|---|---|---|---|
| R lat. (us) | 40 | 22 | 12 |
| R bw (GiB) | 3.5 | 7 | 14 |
| W lat. (us) | 60 | 25 | 13 |
| W bw (GiB) | 2.5 | 5.5 | 12 |

**Figure 2.** *R*ead and *W*rite of three types of SSDs.

size of 256B, the gap with the low-speed SSD is almost closed. However, the gap with the high-speed SSD is still over 2X.

The significant performance gaps suggest that the overhead of the NVMe interface can be largely exposed on the critical path of serving small requests and becomes a performance bottleneck. With performance gaps at this scale, the vision and practice of offloading KV store behind the NVMe interface and making it a decoupled service would be foiled as few can trade so much performance loss for architectural advantages. The issue is genuine and widespread as small KVs are common in various production workloads. For example, Facebook had reported that in its Memcached KV pool dedicated for storing user-account statuses, all values are of 2 bytes. In its general-purpose pool, almost 40% of the total requests are for values no more than 11 bytes [1]. In a replicated pool, the mean value size is 66 bytes [12]. In Twitter's KV workloads, each tweet under compression has only 362 bytes with only 46 bytes of text as the value [5]. The issue is further aggravated when the KV store itself becomes increasingly fast with the increase of SSD speed and employment of new devices such as persistent memory.

To address the issue and pave the road to effectively offloading the KV-store as an independent service via NVMe, we need to minimize the number of small requests. Accordingly, we propose SAKER (Software Accelerated Key-value sERvice), a software framework that facilitates offloading a KV store off the host and behind the NVMe interface with minimal performance loss even with frequent small requests. SAKER is designed as a lightweight software layer that requires no hardware modifications, works with standard NVMe-KV interfaces, and can accelerate any KV-SSD. While SAKER uses two conventional techniques (batching small writes in a write buffer and serving small read requests from a read cache on the host side) it addresses some unique challenges, including lock contention and user-directed data persistency (sync) at the write buffer, and work concurrency in the KV SSD. The novelty of this work lies in its transparency and generality: it enables an efficient NVMe KV interface without requiring application modifications or restricting its benefits to specific applications.

## 2 Background and Related Work

The newly ratified NVMe-KV Command Set has been highlighted with benefits such as removing a translation layer, support of computational storage, removing space provisioning overhead, and consistent key space across multiple devices [10]. NVM Express, or NVMe, was proposed to extend the PCIe communication protocol for external high-speed non-volatile memory devices, such as SSDs, and to replace traditional SATA and SCSI interfaces for block storage devices. More recently, the NVMe Command Set Specifications further support new features introduced in the storage devices such as ZNS [13], FDP [4], and key value store. These supports make the capabilities in an SSD directly available to the host software. Moreover, NVMe commands can also be sent over the network with NVMe-over-Fabric (NVMe-oF), which allows NVMe KV service to be the target of NVMe-oF. While direct access of capabilities in a (remote) device simplifies the operation logic and reduces software overhead at the host, it exposes the potential performance risk on the path between the service requester and the service provider.

There have been some studies on KV SSDs. However, few directly address the interface overhead issue. One such approach introduces compound commands and ask programmers to use them to aggregate small KV requests into one command [8]. Apparently, it's not an ideal option requiring a program's I/O operations to be restructured. Another work for developing a KV SSD simply assumes 4KB values to amortize the overhead [7]. While there are other works proposed to optimize KV SSDs, their goals differ significantly from addressing interface overhead. KAML [6] and KV-CSD [14], for example, offload substantial functionality to the storage device. KAML offers native transactional support with fine-grained locking through custom hardware and includes a host-side cache. But its primary focus is on providing transactional semantics. KV-CSD adopts a computational storage model by embedding an LSM-tree-based KV store on an ARM SoC within the SSD, fundamentally altering the storage architecture. Neither conforms to standardized interfaces. Dotori [3], on the other hand, addresses the performance gap between KV SSDs and block SSDs by proposing a novel B+-tree index, the OAK-tree, tailored for KV SSDs. While Dotori incorporates host-side buffering through an index update buffer and a KV cache, these components are tightly coupled with its indexing scheme and specifically designed for its out-of-order, append-only structure. As such, Dotori functions as a full-fledged key-value store on the host rather than a general-purpose acceleration layer for applications. By maintaining buffer/cache on the host, Dotori improves the KV store's performance. However, its primary motivation is not to improve KVSSD performance, but rather to enhance KVSSD functionality and better leverage its capabilities.

A recent work, ByteExpress, identifies page-based DMA data copying as a major source of inefficiency in the NVMe interface for small data transfers [15]. Specifically, data transfer over PCIe between host memory and device DRAM occurs in units of 4KB page. For small KV items, this leads to significant transfer amplification, contributing to the interface bottleneck. ByteExpress addresses this by enabling fine-grained data delivery over PCIe: it embeds data directly within the NVMe request by appending 64-byte submission queue entries to the NVMe command. While this technique improves small write performance, its additional overhead of handling multiple queue entries can offset the gain, especially when transferring moderately large data items.

Compared to these approaches that require custom hardware, complete application restructuring, or new protocol implementation, SAKER offers a lightweight software layer that specifically targets NVMe-KV interface overhead. It provides transparent acceleration for both read and write requests—regardless of data size—without requiring intrusive changes to hardware or application logic.

To understand the impact of the NVMe-KV interface on the service path, we need an NVMe SSD device and the ability to flexibly configure its parameters and features. However, at this early stage of the NVM-KV standard, few NVMe-KV-based storage devices available for the investigation. Furthermore, this study requires exploring the design space in the SSD extensively. Therefore, we choose to use an NMVe SSD emulator (NVMeVirt [9]). NVMeVirt is implemented on top of the PCIe device emulation. Its emulated device presents itself as real to the rest of the OS and other devices. The host can set PCI's control block and place operations in the NVMe queues to carry out NVMe operations. Our implementation leverages the user-level SPDK NVMe driver to access the device directly, bypassing the kernel. Consequently, SAKER's host-side software layer is architecturally designed to be portable and fully compatible "as-is" with any real NVMe-KV SSD that conforms to the standardized NVMe-KV Command Set. The NVMe message queues in the driver are the primary mechanism to exchange requests and results. There are two types of queues: submission queue and completion queue. Each submission queue is paired with a completion queue, forming a queue pair.

## 3 The Design

There are two objectives in the design. One is to port real-world KV stores into the NVMe KV SSD emulator to reveal performance implications of the NVMe KV interface on next-generation KV SSDs. The second is to propose efficient methods to prevent the NVMe interface from becoming a performance bottleneck in access to NVMe KV SSDs.

### 3.1 The System Architecture

There are two architectures considered in this study, which are the HostKV and DiskKV, as illustrated in Figure 3. HostKV is the currently dominant KV store structure, where KV store runs in the application context without NVMe involvement. Only requests on data blocks are sent to the NVMe disk. However, in DiskKV, the application is decoupled from the KV store with an NVMe KV interface between the two. To reduce small requests to reach the SPDK NVMe driver queues, SAKER's buffer cache is placed between the applications and the driver. In the NVMeVirt emulated SSD, a dispatcher thread retrieves commands from submission queues, determines its target completion time according to a media access performance model, and delivers them to one of the FIFO I/O queues in a round-robin manner (Figure 3). There is an I/O thread dedicated to each I/O queue. The I/O thread is responsible for retrieving the commands from its I/O queue and executing them in the KV store on the device. Compared with the KV store in HostKV, the KV store in DiskKV doesn't directly receive requests from the application threads. Instead, it receives requests from the I/O threads in the device.

### 3.2 KV Stores in an in-Kernel Emulator

The NVMeVirt KV SSD emulator [9] is a Linux kernel module in which a hash-table-based KV store is implemented to organize the KV items on the flash and index them in the memory. However, this implementation only provides a primitive KV store that doesn't meet requirements of the major use scenarios for this study. For example, many applications use range queries, which cannot be efficiently supported by hash-table-based KV stores. As the KV store resides on a block device, sophisticated space management is required to accommodate variably-sized values with minimal write amplification. Many KV stores have been developed with extensive optimizations, such as RocksDB. To understand the impact of state-of-the-art KV stores on the NVMe interface, we port more KV stores into the emulator.

There are two challenges in the porting. One is that most KV stores are developed as user-level applications relying on a file system to manage disk space. Apparently, a full file system is not expected to run within a storage device (though its data structures may reside there), which usually provides a block or key address space, rather than a hierarchy of directories and files. Second, the source code of a KV store application usually has dependencies on many user-level libraries. To address these issues, we reposition NVMeVirt into the user level and accordingly support access of the emulated device via an SPDK user-level NVMe driver. To facilitate disk space management, a file system is needed between a KV store and the SSD medium. However, we do not assume a full-fledged OS and its file system running in the SSD. Instead, we use a simplified in-memory file system [20] as the store's underlying file system. For a fair comparison, we also use this file system for HostKV in the experiments. It should be noted that "in-memory" means the data are stored in memory; however, we also write the data the disk emulator to simulate the flash write cost.

In addition to a hash-based KV store and RocksDB, we ported a COW-friendly B-tree-based KV store [21] to the emulator. Meanwhile, for comparison, they are also implemented at the host and access the NVMeVirt-emulated disk

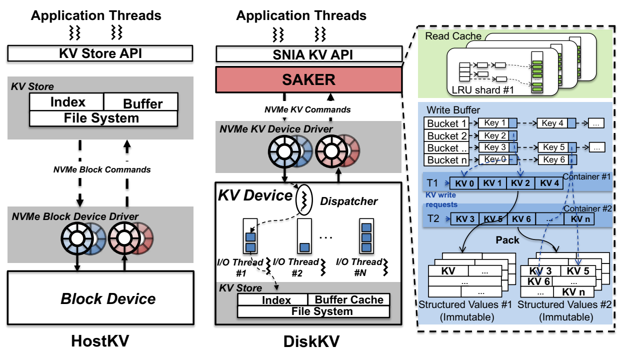

Figure 3. The HostKV and DiskKV architectures

via its NVMe block interface. For each of these KV stores, both its DiskKV and HostKV implementations use SPDK user-level driver, NVMe interface, and the same set of performance parameters in the emulation for a fair comparison.

## 3.3 SAKER's Write Buffer at the Host

SAKER is designed as a software support to close the potential performance gap between HostKV and DiskKV due to small requests across the NVMe KV interface. The root cause of the gap is that the operational cost at the NVMe KV interface cannot be amortized with a small request. In contrast, the NVMe block interface in the HostKV enforces a large minimal access unit (e.g., 4KB), amortizing the cost. To address this, SAKER has two methods: (1) making write requests large enough with a write-back buffer and (2) serving read requests in a cache before they cross the interface.

A KV store has its in-memory buffer cache. Some KV stores set up a relatively large write buffer. For example, LSM-tree-based stores, such as RocksDB, use the buffer (MemTable) to generate on-disk SSTable, which is often of tens of megabytes. For the B-tree-based KV store, its write-back buffer is to reduce the write amplification on the block device. A large buffer is needed to accommodate random write workloads. SAKER additionally sets up a write-back buffer at the host side. The purpose is to collect write requests and batch them into a large one. For this purpose, SAKER's buffer is relatively small. However, its operations are highly latency-sensitive. Without a careful design, its overhead may overtake the cost at the NVMe interface, which would defeat the purpose of improving the interface's efficiency.

A major source of performance loss in the design is lock contention. To this end, we collect multiple write requests into a write request container. The container is considered full when the number of its KV requests reaches a batching limit, the total request size reaches the container's capacity, which is 4KB, or a sync command is received, whichever comes first. There are two potential lock-contention scenarios. One is that multiple threads simultaneously place their write requests to a container. The other is that multiple threads simultaneously send their commands to the same SPDK's submission queue. Both scenarios require serializing access to either of the data structures. To avoid this performance loss, SAKER dedicates a request container and a SPDK queue pair to an application thread. When a container is full, SAKER assembles its requests into one write NVMe KV command. All its constituent requests are packed into a structured value, which is the value of the command.

When multiple commands from the same thread are placed in a submission queue pending for service, the request buffering and batching strategy essentially enables asynchronous writes. For high performance, SAKER temporarily leaves the data in the volatile memory. An application thread wanting its data to be immediately persisted can issue a sync command to make all its currently pending writes in the container and submission queue be serviced before an acknowledgment is returned. This is the same design choice adopted by most file systems on their buffer cache.

All the KV data in the pending write requests are visible to the following read requests. If there are multiple writes about the same key, SAKER returns the latest one. We use a hash table to organize the locations of the write requests in the containers and the structured values associated with the pending commands in the submission queues (see "DiskKV" in Figure 3). When a thread sends a new write request or an NVMe command is completed, the hash table needs to be updated. To minimize the lock contention at the hash table shared by all threads, SAKER sets up a lock for each hash bucket. As the number of buckets is much larger than the thread count, a read request can access the values in the write buffer with little lock contention. By using a hash table, range queries cannot be efficiently conducted. If a range query is a synchronous one, the application thread needs to issue a sync command before the query. Otherwise, the data is not included in the service of the range query.

With the addition of a host-side write buffer, the KV-serving stack includes two separate buffering layers for write requests. While this may seem redundant, it is both necessary and poses little concern for space efficiency. First, the two buffers serve distinct purposes: SAKER's host-side buffer batches small NVMe-KV commands to amortize interface overhead, whereas the device-side buffer, managed by an offloaded KV store (e.g., RocksDB's MemTable), organizes data into larger, sequential write structures (e.g., SSTables) to optimize flash storage and minimize write amplification. Second, the host-side buffer is much smaller—typically a few megabytes compared to hundreds of megabytes for the device-side buffer—making any redundancy between them negligible in terms of space consumption.

## 3.4 SAKER's Read Cache at the Host

Different from the write buffer, the read cache needs a large space to capture a likely large working set. A KV store usually maintains a read cache to minimize access to the block

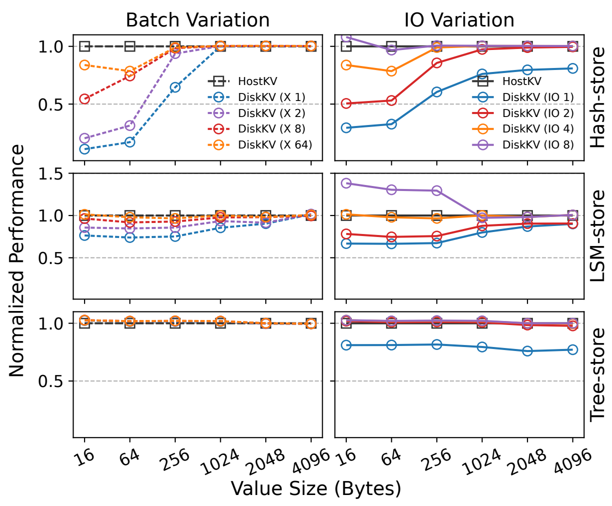

**Figure 4.** Normalized write throughput of Hash-store, LSM-store, and Tree-store on HostKV and DiskKV with different batch sizes (the left column of graphs, where **(X *n*)** indicates batch size *n*) and I/O thread counts (the right column of graphs, where **(IO *k*)** indicates number of I/O threads *k* when the batch size is 4) under random writes. Throughput is normalized to its that with HostKV.

device. SAKER needs a read cache at the host side to minimize (small) KV read requests crossing the NVMe interface. When SAKER services a read request from its cache, it also eliminates a read from the disk medium. Therefore, the read cache of the KV store can be reduced or even removed. In other words, the cache is repositioned to the host side and managed by SAKER. Being placed in the front of the request service path, the cache helps reduce both the overhead of crossing the NVMe interface and the cost of accessing the disk medium. For a fair comparison, we use the implementation of RocksDB's KV read cache (the row cache) as SAKER's read cache, including its LRU policy, with a new admission control: only KV items smaller than a threshold are admitted into the cache, because caching large KV items is not necessary (as the NVMe overhead can be well amortized) and is also too expensive (as it consumes excessive cache space).

## 4 Evaluation

We run experiments to understand the performance impact of the NVMe interface overhead on a KV store in the KV SSD. We also evaluate the effectiveness of host-side write batching and read caching to amortize or remove the overhead with three different KV stores as well as varying KV item sizes, batching sizes, number of I/O threads in the disk, cache sizes, and access locality. This extensive study will help to pinpoint the pain points of the NVMe-based KV store architecture and reveal the extent to which they can be alleviated. In the evaluation, we set the four performance-related parameters in the NVMeVirt emulator as follows. The read latency and bandwidth are 22us and 7GiB, respectively. The write latency and bandwidth are 25us and 5.5GiB, respectively. The key size is 8B. The three KV stores are a hash-table-based KV store (*Hash-store*), an LSM-tree-based KV store (RocksDB [19]) (*LSM-store*), and a B+-tree-based KV store [21] (*Tree-store*). Experiments are conducted on a server with dual Intel Xeon E5-2695 v4 18-core processors (2.10 GHz) with Hyper-Threading enabled and 256 GB DRAM. To ensure fair performance isolation and minimize interference, CPU cores are partitioned between host-side software and the emulated device by assigning them to different NUMA nodes.

**Write Throughput** In the experiment, four application threads send 20 million write requests. We first fix the I/O thread count at 4 and vary batch size from 1 to 64 and value size from 16B to 4KB. As shown in the left three graphs in Figure 4, the performance gap between DiskKV and HostKV is most pronounced with small value sizes and lighter-weight KV stores, as the gap can be as large as 10X (on Hash-store). With a light-weight KV store, the NVMe interface is more likely to become the bottleneck. This 10X performance improvement clearly demonstrates that small KV requests disproportionately amplify the cost of the NVMe-KV interface. Without large KV commands to amortize its overhead, the bottleneck is materialized. With batching of enough requests in a command, the gaps can be (largely) closed. In contrast, with a heavy-weight KV store (Tree-store under random writes), the gap doesn't show up. With a large batch size, DiskKV's throughput closely matches that of HostKV, indicating that SAKER introduces minimal operational overhead compared to the execution cost of a host-side KV store.

In another experiment, we fix the batching size at 4 and change the I/O thread count from 1 to 8 in DiskKV. Due to the decoupling of application and KV-store software, DiskKV can independently adjust the processing power, quantified by number of I/O threads, each running on its own core. With more I/O threads, DiskKV's throughput can be higher than HostKV when performance of the KV store is bottlenecked by the CPU, such as RocksDB (an LSM-store) with high CPU demand for its intensive compaction operations. Arguably, the computing power in a storage device is not supposed to be as high as, or even higher than that on the host. However, this experiment result is still relevant and encouraging, as the device behind an NVMe interface is not limited to an SSD device. It can also be a KV service provider hosted on one or a cluster of servers, including disaggregated KV store via NVMe KV interface.

**Read Throughput** In the experiment, we choose LSM-store as the KV store. The workload is to use four read threads to randomly read over a 500MB data set with zipfian key access distribution and different cache sizes, value sizes, and skewness (*s*). The selected read cache sizes are 0 MB, 128 MB, and 256 MB, where 0 MB indicates that the read cache is disabled. As shown in Figure 5, the throughput gaps are

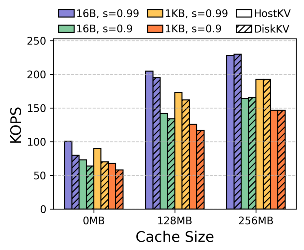

**Figure 5.** Read Thruput

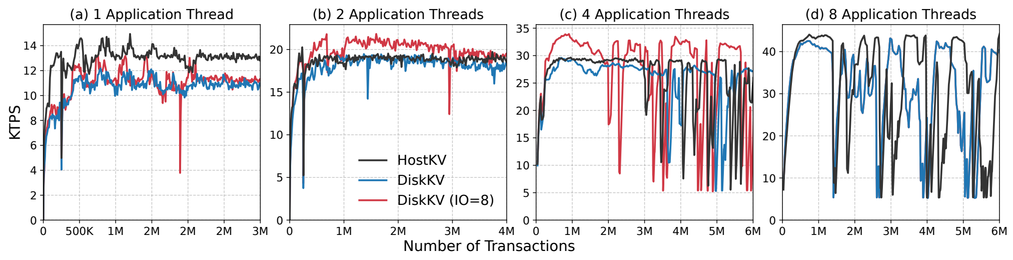

**Figure 6.** Comparing HostKV and DiskKV under TPC-C on LSM-store (512MB cache)

generally moderate, which is expected. Any miss in the cache leads to a block read in the disk medium. If the value is small (such as 16B), the read amplification on the block device dominates the service time and the NVMe interface is not a major bottleneck. If the value is large (such as 1KB), the overhead at the interface can be amortized. Without the read cache (cache size is 0MB), the gap between HostKV and DiskKV is around 17-25%. With large caches (128MB or 256MB), the gap reduces to less than 5% across the cache sizes and zipfian skewness. The small gaps reveal the advantage of repositioning the read cache. The KV SSD vendors therefore are suggested to minimally provision the read cache.

However, the write buffers on the host and device sides serve distinct purposes. The host-side buffer in SAKER is used to batch small NVMe-KV commands, helping to amortize interface overhead. In contrast, the write buffers of an offloaded KV store (e.g., RocksDB's MemTable) are designed to prepare larger, sequential write structures (SSTables) for on-flash storage and to reduce write amplification. The host-side buffer is significantly smaller than the offloaded KV store's buffer—a few megabytes versus hundreds of megabytes. As a result, any redundancy between the two has minimal impact on overall space efficiency.

**The TPC-C Benchmark** TPC-C is a benchmark mixed with read/write requests to evaluate performance of OLTP databases [22] on LSM-store. In the experiment, we use two of its five types of transactions (New Order and Payment), which account for nearly 90% of TPC-C's workload. A major operation is to process an order line, which reads from the ITEM and STOCK tables and writes into the ORDER_LINE table. In the workload, read and write requests account for about 60% and 40%, respectively. Figure 6 shows throughput during a 300-second execution with different application thread counts. The I/O thread count can be either the same as that of the application threads ("DiskKV" in the graph) or fixed at 8 ("DiskKV (IO = 8)"). The batching size is 64 and the cache size is 512MB. It is observed that the throughput curves fluctuate during request processing, with the fluctuations becoming more pronounced as the number of threads increases. This behavior is attributed to compaction operations, which are inherent to any LSM-based KV store. With more threads the write intensity increases, triggering compactions more frequently and resulting in greater throughput variability.

With only one thread, the throughput gap cannot be closed because of the read/write dependency. A set of write requests can be issued only after their contingent read request is completed. Although the sync command is not used—which would otherwise prevent batching of write requests before and after it—concurrency and batching are still limited under single-threaded execution. When there are two or more application threads, the gap can be closed. Furthermore, when there are more I/O threads, the increased processing power on RocksDB for its compaction leads to higher throughput of DiskKV than HostKV (Figures 6 (b) and (c)). With a 512MB cache, the hit ratio is 95%. If we reduce it to 32MB, the ratio becomes 76%. With two application threads, the throughput gap is about 15% due to choked writes caused by their dependency on reads. With 8 application threads, the increased concurrency helps almost close the gap (less than 1%).

## 5 Conclusion and Future Work

This work experimentally reveals the impact of the NVMe KV interface on KV store workload and proposes SAKER to close this performance gap. SAKER employs a read cache, write buffer, and batching to allow NVMe KV stores to perform competitively to local on-host KV stores. It presents a potential solution to cost-effectively introduce the NVMe KV command set into next-generation KV products.

It is acknowledged that while the NVMe KV Command Set is designed to provide a key-value interface at the device level, it does not natively support advanced features such as transactional operations or range queries across multiple KV items. Currently, applications must implement these capabilities at the host side using only the basic PUT, GET, and DELETE commands. As future work, we plan to enhance SAKER's capabilities by either enabling direct support for these features or facilitating their optimized implementation at the application level.

## Acknowledgments

We thank the anonymous reviewers for their valuable feedback. This work was supported by a generous gift from VMware, Inc., and partially by the U.S. National Science Foundation under Grant CCF-2313146.

## References

[1] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. 2012. Workload analysis of a large-scale key-value store. *SIGMETRICS Perform. Eval. Rev.* 40, 1 (June 2012), 53–64. doi:10.1145/2318857.2254766

[2] Bigtable [n. d.]. Bigtable: Fast, Flexible NoSQL. https://cloud.google.com/bigtable.

[3] Carl Duffy, Jaehoon Shim, Sang-Hoon Kim, and Jin-Soo Kim. 2023. Dotori: A Key-Value SSD Based KV Store. *Proc. VLDB Endow.* 16, 6 (Feb. 2023), 1560–1572. doi:10.14778/3583140.3583167

[4] Flexible Data [n. d.]. Flexible Data Placement: State of the Union. https://nvmexpress.org/wp-content/uploads/FMS-2023-Flexible-Data-Placement-FDP-Overview.pdf.

[5] How much text [n. d.]. How much text versus metadata is in a tweet? https://gist.github.com/brendano/1024217.

[6] Yanqin Jin, Hung-Wei Tseng, Yannis Papakonstantinou, and Steven Swanson. 2017. KAML: A Flexible, High-Performance Key-Value SSD. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 373–384. doi:10.1109/HPCA.2017.15

[7] Yangwook Kang, Rekha Pitchumani, Pratik Mishra, Yang-suk Kee, Francisco Londono, Sangyoon Oh, Jongyeol Lee, and Daniel D. G. Lee. 2019. Towards building a high-performance, scale-in key-value storage system. In *Proceedings of the 12th ACM International Conference on Systems and Storage* (Haifa, Israel) *(SYSTOR '19)*. Association for Computing Machinery, New York, NY, USA, 144–154. doi:10.1145/3319647.3325831

[8] Sang-Hoon Kim, Jinhong Kim, Kisik Jeong, and Jin-Soo Kim. 2019. Transaction Support using Compound Commands in Key-Value SSDs. In *11th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 19)*. USENIX Association, Renton, WA. https://www.usenix.org/conference/hotstorage19/presentation/kim

[9] Sang-Hoon Kim, Jaehoon Shim, Euidong Lee, Seongyeop Jeong, Ilkueon Kang, and Jin-Soo Kim. 2023. NVMeVirt: A Versatile Software-defined Virtual NVMe Device. In *21st USENIX Conference on File and Storage Technologies (FAST 23)*. USENIX Association, Santa Clara, CA, 379–394. https://www.usenix.org/conference/fast23/presentation/kim-sang-hoon

[10] Bill Martin. 2024. NVMe® Key Value Command Set Provides the Key to Storage Efficiency. https://nvmexpress.org/nvme-key-value-command-set-provides-the-key-to-storage-efficiency/.

[11] MySQL [n. d.]. MySQL: The world's most popular open source database. https://www.mysql.com/.

[12] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. 2013. Scaling Memcache at Facebook. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation* (Lombard, IL) *(nsdi'13)*. USENIX Association, USA, 385–398.

[13] NVMe Zoned [n. d.]. NVMe Zoned Namespaces (ZNS) Command Set Specification. https://nvmexpress.org/specification/nvme-zoned-namespaces-zns-command-set-specification/.

[14] Inhyuk Park, Qing Zheng, Dominic Manno, Soonyeal Yang, Jason Lee, David Bonnie, Bradley Settlemyer, Youngjae Kim, Woosuk Chung, and Gary Grider. 2023. KV-CSD: A Hardware-Accelerated Key-Value Store for Data-Intensive Applications. In *2023 IEEE International Conference on Cluster Computing (CLUSTER)*. 132–144. doi:10.1109/CLUSTER52292.2023.00019

[15] Junhyeok Park, Junghee Lee, and Youngjae Kim. 2025. ByteExpress: A High-Performance and Traffic-Efficient Inline Transfer of Small Payloads over NVMe. In *Proceedings of the 17th ACM Workshop on Hot Topics in Storage and File Systems* (Boston, MA, USA) *(HotStorage '25)*. Association for Computing Machinery, New York, NY, USA, 114–121. doi:10.1145/3736548.3737837

[16] PCIe 3.0 [n. d.]. Samsung 970 PRO PCIe 3.0 NVMeSSD. https://www.techpowerup.com/ssd-specs/samsung-970-pro-512-gb.d54.

[17] PCIe 4.0 [n. d.]. Samsung 980 PRO PCIe 4.0 NVMe SSD. https://www.techpowerup.com/ssd-specs/samsung-980-pro-1-tb.d47.

[18] PCIe 5.0 [n. d.]. crucial-t700 PCIe 5.0 NVMe SSDs. https://www.techpowerup.com/ssd-specs/crucial-t700-pro-4-tb.d1857.

[19] RocksDB [n. d.]. RocksDB | A Persistent Key-value Store. https://rocksdb.org/.

[20] Rocksdb [n. d.]. RocksDB Mock File System. https://github.com/facebook/rocksdb/blob/main/env/mock_env.h.

[21] Ohad Rodeh. 2008. B-trees, shadowing, and clones. *ACM Trans. Storage* 3, 4, Article 2 (Feb. 2008), 27 pages. doi:10.1145/1326542.1326544

[22] TPCC [n. d.]. TPC-C is an On-Line Transaction Processing Benchmark. https://www.tpc.org/tpcc/.

[23] NVM Express Workgroup. 2024. NVM Express Key Value Command Set Specification, Revision 1.1. https://nvmexpress.org/specifications/.