Transaction Support using Compound Commands in Key-Value SSDs

Sang-Hoon Kim
Ajou University

Jinhong Kim, Kisik Jeong
Sungkyunkwan University

Jin-Soo Kim
Seoul National University

Abstract

Recently proposed key-value SSD (KVSSD) provides the popular and versatile key-value interface at the device level, promising high performance and simplified storage management with the minimal involvement of the host software. However, its I/O command set over NVMe is defined on a per key-value pair basis, enforcing the host to post key-value operations to KVSSD independently. This not only incurs high interfacing overhead for small key-value operations but also makes it subtle to support transactions in KVSSDs without a software support.

In this paper, we propose *compound commands* for KVSSDs. The compound command allows the host to specify multiple key-value pairs in a single NVMe operation, thereby effectively amortizing I/O interfacing overhead. In addition, it provides an effective way for defining a transaction comprised of multiple key-value pairs. Our evaluation using a prototype KVSSD and an in-house KVSSD emulator shows promising benefits of the compound command, with improving the performance by up to 55%.

1 Introduction

Recent years have witnessed the drastic changes and evolvement of storage systems. New storage media based on the state-of-the-art semiconductor technologies have been introduced, and storage devices built with those emerging media are arriving on the market [10–12, 20]. To fully leverage their high performance and unique characteristics, it is required to renovate the storage stack and components. Examples include bypassing the deep operating system layers [14, 22, 26], fusing main memory with secondary storage [5], making storage devices smarter [9, 15], and so forth. One of noticeable directions is to reorganize I/O architecture and improving the host-device interface. Specifically, NVM Express (NVMe) [18] allows storage devices to be attached to the host bus via the PCI Express (PCIe) interface. This approach enables storage devices to interface with the host at high bandwidth and low latency which match those of modern storage media.

In the meantime, key-value stores become one of the most popular software services to build and operate large-scale data-intensive applications and services [2, 4, 6, 8, 17]. Due to their simple yet effective interface, many enterprise services have been adopting the key-value abstraction in their systems. For instance, Amazon S3 [2] has been the foundation of many IT services with its scalability, data availability, security, and performance. Memcached [8] is to accelerate services by caching operation results via the key-value interface. Motivated by these software key-value stores and services, key-value SSD (KVSSD hereafter) attempts to provide the key-value store service at a device level [21]. By completely replacing deep software stack with off-the-shelf hardware, KVSSD can directly respond to data requests from an application with minimal involvement of the host software, thereby increasing performance and simplifying storage management [21].

In spite of those promises, we argue that the host-KVSSD interface is yet to be improved. KVSSD interfaces with the host through the *KVSSD command set* which is extended from the NVMe command standard [27]. The current commands are defined on a per key-value pair basis. Thus, the host has to request one key-value operation at a time to KVSSD, and KVSSD should process each key-value pair individually. This becomes problematic when the size of keys and values are small, which is commonly observed from many real key-value store services [3]. Worse, it is difficult to define a relationship among multiple key-value pairs since the current commands lack of handling multiple key-value pairs collectively.

In this paper, we strive to break the limitation of the current KVSSD interfaces. We propose a *compound command* that allows multiple key-value operations to be included in a single NVMe command. This enables multiple small key-value operations to be coalesced into a single NVMe command, thereby effectively amortizing interfacing overhead over NVMe. Furthermore, we can intuitively define a sophisticated semantic on a set of key-value pairs. Examples include transaction support, group prefetching, and so forth.

We evaluate the performance benefit of compound commands using a prototype KVSSD and an in-house KVSSD emulator. Evaluation result shows compound commands can effectively amortize the interfacing overhead, reducing per-operation latency by up to 55%.

The rest of this paper is organized as follow; Section 2 explains the proposed the compound command and its APIs. Section 3 compares the benefit of compound commands over the current per-tuple operations, and Section 4 concludes this paper.

2 Compound Commands Design

2.1 Motivation

Exchanging multiple small I/O messages incurs higher overhead than transferring large data between a host and a peripheral device. Thus, many I/O systems try to amortize the interfacing overhead by merging small I/O requests into a single I/O command and transferring the requests in bulk. For example, the eMMC standard [13] defines the *packed command*; instead of exchanging I/O requests and results individually, the host (more specifically I/O schedulers and/or device drivers) collects multiple eMMC operations (i.e., writing data on specified blocks), and builds a packed command that coalesces multiple operations in a single eMMC command. When the host posts the packed command, the eMMC device fetches the data from the host in bulk and then internally processes each operation in batch. In this way, eMMC devices can leverage their I/O bandwidth to handle small I/O requests efficiently.

The same story can be applicable to the KVSSD interface. The KVSSD command set [27] extended from the standard NVMe command set supports five primitive key-value operations which are STORE, RETRIEVE, DELETE, EXIST, and ITERATE. Specifically, STORE saves a value for a given key. The stored value can be either accessed with the RETRIEVE command or deleted with the DELETE command using the key. The EXIST command is for checking whether KVSSD stores a value for the given key. The ITERATE command is for querying key-value tuples having a specified prefix in keys. All those commands are defined so as to specify one key or one key-value pair in the NVMe command message. Thus, even though the NVMe interface standard features low latency and high bandwidth, KVSSD may not operate at its full performance but spend considerable execution time on frequent interfacing with the host. In addition, it is difficult to support an operation that is defined or has to be applied to multiple key-value pairs all together; each key-value operation is carried out independently using those per key-value commands.

Based on the observation, we propose *a compound command*. The compound command complements the current KVSSD command set by allowing multiple key-value operations to be encoded in a single NVMe command. Moreover, we found that using the compound command makes it easy to define a transaction over multiple key-value operations. The rest of this Section explains how the current command set is extended and how the compound command can be used to support transactions in KVSSD.

2.2 Compound Command Formats

The original KVSSD commands comply the 64-byte NVMe command format. Key-value operations with small keys are frequently observed from real workloads [3]. To efficiently handle this common case, a key can be inlined in the command (i.e., directly written on the NVMe command) if the key length is shorter than or equal to 16 bytes. When the key is longer than 16 bytes, the key is specified in the command *indirectly* using two fields each of which contains the memory location of the key and the key length. When KVSSD receives a command from a host, it firstly checks whether the key is inlined. If that is the case, KVSSD accesses the inlined key directly from the command. Otherwise, KVSSD fetches the key from the host's main memory. This memory access across device boundaries can impose an additional overhead compared to accessing inlined keys. Unlike keys, values are always specified in the command indirectly with a tuple comprised of the memory address of the value and its length. KVSSD fetches the value from or pushes the value to the host's main memory through memory operations like DMA.

We extended the original KVSSD command set to support the compound command. Overall, compound commands use the same NVMe command format of the original KVSSD commands but use the key and value fields for other meanings. The fields originally used for holding an inlined key are used for specifying an optional *identification number (ID)* of the key-value operations grouped in the compound command. The ID is limited to 16-bytes long to be inlined in the NVMe command, and can be utilized to (but not limited to) specify a transaction ID or a group ID for the given key-value list.

The fields originally used for specifying the value are now used for specifying the location of the *command payload* that contains multiple key-value pairs in a self-explanatory format. For key-value operations requiring both key and value (e.g., STORE), key-value pairs are represented on a contiguous memory buffer in (key length, key, value length, value) format which are preceded by the total number of pairs. For key-value operations requiring only keys (e.g., RETRIEVE, DELETE, and EXIST), the keys are listed on a contiguous memory buffer in (key length, key) format which are preceded by the total number of keys. Figure 1 illustrates the layout of the command payload for an extended STORE compound command which

We opt for this command payload layout to minimize the number of cross-device memory operations. Building

0x00	3 (number of tuples)				8 (key1 length)				1	2	3	4	5	6	7	8
0x10	16 (val1 length)				1	2	3	4	5	6	7	8	9	10	11	12
0x20	13	14	15	16	7 (key2 length)				1	2	3	4	5	6	7	
0x30	10 (val2 length)				1	2	3	4	5	6	7	8	9	10		
0x40	23 (key3 length)				1	2	3	4	5	6	7	8	9	10	11	12
0x50	13	14	15	16	17	18	19	20	21	22	23		512 (val3 length)			
0x60	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
…					…				…							

Figure 1: The layout of a STORE compound command payload which contains three key-value pairs. In the NVMe message for compound commands, the value field specifies the memory location and the total length of the payload. KVSSD can fetch the entire command payload with a single memory operation.

a packed command payload involves one memory copy for each key-value pair. To avoid the memory copy overhead, one might suggest to use a vector specifying the address of each key and value. However, to fetch these operands, KVSSD should perform multiple cross-device memory operations since the memory operation can be performed on a contiguous memory region but key-value pairs are likely to be dispersed on the host's main memory. IOMMU might provide KVSSD with a virtually contiguous memory region over the key-value pairs, however, it requires memory mapping and unmapping for each key and value, which incurs non-trivial overhead. This becomes even more problematic when keys and values are small and the number of key-value pairs grouped in a compound is huge. On the other hand, KVSSD can fetch the entire key-value pairs with a single memory operation by using the packed command payload, and by which the memory copy overhead can be amortized.

2.3 Transaction Support and User API

The current KVSSD commands are defined to be processed *independently of each other*. Thus, even though a set of operations are consecutively requested to KVSSD, they can be interleaved by other operations. This *independency among operations* makes KVSSD unable to guarantee the *atomicity* and *isolation* properties of transaction. Specifically, KVSSD may hold a partially processed state when the host crashes in the middle of processing transaction operations. Moreover, when an operation is interleaved between the transaction operations, it can not only interrupt the ongoing transaction but also access an intermediate state of the transaction, which violates the isolation property. In this sense, the current KVSSD cannot fully realize the ACID properties for transaction without software support, which opposes to the key idea of the hardware-based key-value store to make I/O path short.

The compound command can complement the current KVSSD command set to support transaction. The compound command enables the host to define a transaction over multiple key-value operations easily; the host can collect key-value operations, compose a compound command with the operations, mark it as a transaction, and post the compound command to KVSSD. KVSSD should ensure the ACID properties while processing the operations in the transaction. To this end, KVSSD can disallow other operations to be interleaved between transaction operations. At the same time, KVSSD leverages traditional write-ahead logging (WAL) techniques to atomically update multiple key-values; write values, metadata updates, and key index updates on a logging area first, and then atomically update key index and metadata all together. These approaches allow KVSSD to guarantee the atomicity and isolation properties of transactions. Also, the durability property can be easily provided by properly and safely storing data in KVSSD.

Providing the consistency property in KVSSD is, however, subtle. For now, KVSSD can provide limited consistency by using compound commands; KVSSD can check whether each operation does not violate any consistency restriction of the system (i.e., the value should not be NULL). KVSSD can proceed to process the transaction only if all operations satisfy the conditions. Otherwise (e.g., if a STORE operation tries to save NULL as the value), KVSSD can reject the transaction. This level of consistency was sufficient to run Ceph on top of KVSSD with compound commands as discussed in Section 3, and we expect this will be also true for many simple applications. However, it is not possible to provide complicated consistency that requires read-modify-write operations since the current compound command design disallows to merge operations with different types. KVSSD can overcome the limitation by adopting the similar techniques of Amazon DynamoDB transaction [1, 23] where a user can specify preconditions for write operations and the write operations are performed only if all preconditions are met. We leave this as future work, and please refer to Section 5 for the details of the extension.

We define user APIs for compound commands considering the transaction support. Listing 1 shows a sample code for the user API. Overall, the API is similar to the transaction operations of RocksDB [7]. A user application can get a

```
int ret;
CC_HANDLE *h;

h = begin_compound(CC_TRANSACTION);
ret = retrieve(h, k1, k1_len, v1, &v1_len);
ret = retrieve(h, k2, k2_len, v2, &v2_len);
ret = commit_compound(h);

h = begin_compound(CC_TRANSACTION);
ret = store(h, k1, k1_len, v1, v1_len);
ret = store(h, k2, k2_len, v2, v2_len);
ret = commit_compound(h);
```

Listing 1: User API for compound commands

handle for a compound command as shown in line 4 and 9 of Listing 1. The argument CC_TRANSACTION indicates the handle is for a transaction rather than for a simple collection of operations. After getting the handle, the application can append key-value operations using the handle (line 5-6 and line 10-11). Once a certain type of operation is appended to the handle, the application can only append the same type of operations for the simplicity of implementation and semantic definition (see Section 5 for extension). The compound command library handles the operation appending by building the compound command payload. When the application submits the command (line 7 and 12), the payload is attached to a NVMe command, and posted to KVSSD. When the compound command is marked as a transaction, KVSSD should handle the key-value operations in the compound command as a transaction as we explained above. Otherwise, KVSSD may process each operation independently.

3 Evaluation

3.1 Performance Estimation

We may perform a precise evaluation of compound commands with realistic workload if we had KVSSD that actually understands the compound commands and properly handles enclosed operations. However, it requires KVSSD firmware modification, which was not feasible to us at the time of paper writing. Alternatively, we estimate the performance implication of compound commands from other metrics that we can actually measure from a real KVSSD [21].

Firstly, we measure T_{key}, the key handling time of the KVSSD. We were provided with an alternative firmware of the KVSSD which changes the KVSSD into an ordinary SSD operational on the traditional block interface. Using the block firmware, we compare the time of handling a STORE operation with 16-byte key and 4 KB value to that of handling a single 4 KB block write operation (note that 4 KB is the minimum block size of the device). We find that the key-value operation takes 46.14 us whereas the block operation takes 13.49 us. Since the two configurations are only different in key handling, we can suppose the latency difference (i.e., 32.65 us) as the key handling time. We can observe that key handling takes a significant fraction (about 70.8%) of the key-value operation time, which indicates efficient key handling will be the key technology for enabling high-performance KVSSDs.

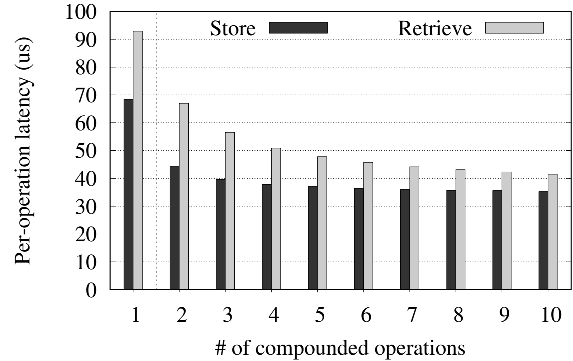

Figure 2: Per-operation latency on various numbers of key-value pairs in a compound command. We calculated the performance of compound commands from the I/O performance of a real KVSSD.

Next, we measure T_{value}, the time to handle values in a compound command. We assume that the compound command-capable KVSSD fetches the command payload similar to the values of ordinary key-value operations and then processes each key-value operation in the payload one after another. We use a 32-byte key and 1,024-byte value configuration, which was the average key-value length of KVCeph workload (See the following subsection for the detail). Thus, a compound command with three key-value pairs will be equivalent to one 16-byte key and 3,196-byte[1] value operation. Since KVSSD will perform one key handling while processing the given key-value operation, we compensate the processing by subtracting T_{key}. Dividing the compensated time with the number of pairs yields the value handling time of the KVSSD.

Figure 2 shows the per-operation time estimated from the obtained T_{key} and T_{value}. Note that '1' is the performance of the original KVSSD command for one 32-byte key and 1,024-byte value operation whereas the rest are the performance of compound commands.

As specified with "Store" and "1", processing one STORE operation with the original KVSSD command takes about 68.4 us. Since the key is longer than 16 bytes, it cannot be inlined in the KVSSD command, and KVSSD should fetch the key with an additional memory operation to the value fetch. Thus, this operation involves (at least) two cross-device memory references, incurring non-trivial overhead. Contrarily, when more than two key-value pairs are packed in a compound command, KVSSD can fetch the entire key-value pairs with a single memory operation. As the number of packed

[1] 3,196 = (4 for key length + 32 for key + 4 for value length + 1,024 for value) $\times$ 3 + 4 for the number of pairs

pairs is increased, the reduced I/O interfacing overhead pays back; it takes only 38.5 us per-operation when ten key-value operations are packed, which is 43.7% latency reduction compared to the original per-operation performance.

RETRIEVE is similarly improved by compound commands. We assume the KVSSD supporting compound command will return the keys and values in bulk in the command payload format presented in Figure 1. Since all results are on a contiguous payload, KVSSD can return the results with one memory operation. The 92.0 us of RETRIEVE operation latency can be reduced down to 41.5 us.

The evaluation result clearly shows that the compound command helps the system to efficiently amortize the interfacing overhead. We believe the similar results will be observed from different key-value length configurations. Also, it is noteworthy that we conservatively imposed the key indexing overhead for each key-value operation. We believe the key indexing overhead can be amortized in practice by batching index update, and if that is the case, compound commands can improve the performance further.

3.2 KVCeph Performance

To evaluate the benefit of the proposed compound command on real workloads, we implemented the compound command on top of our key-value SSD emulator called KVEMU [25]. KVEMU provides a virtual key-value SSD device to a guest OS in a QEMU-provided virtualized environment, similar to FEUM [16]. The guest OS can access a virtual KVSSD using real key-value commands through a virtualized NVMe interface. In addition to providing the functional features of KVSSD, KVEMU emulates the performance characteristics of real KVSSDs. We used the estimated latency explained above to simulate the performance of compound command-capable KVSSD with transaction support (i.e., imposed the per-key handling overhead assuming a strongly conservative processing model).

On top of KVEMU, we setup KVCeph benchmark which is provided as an example of the KVSSD software stack [24]. KVCeph is a variant of Ceph [19] that utilizes the hardware KVSSD in place of its original software key-value store of Ceph. Since KVEMU supports the entire KVSSD command set, we can replace KVSSD with KVEMU without a significant modification.

We attempted to apply the compound command to handle transactions in KVCeph. However, it turned out that the current KVCeph does not handle transaction properly; key-value operations are collected with Ceph transaction APIs, but each key-value operation is processed independently using an individual NVMe command. Thus, transaction properties cannot be guaranteed by the current KVCeph implementation. We believe this case explains the difficulty of defining transaction over multiple independent key-value operations.

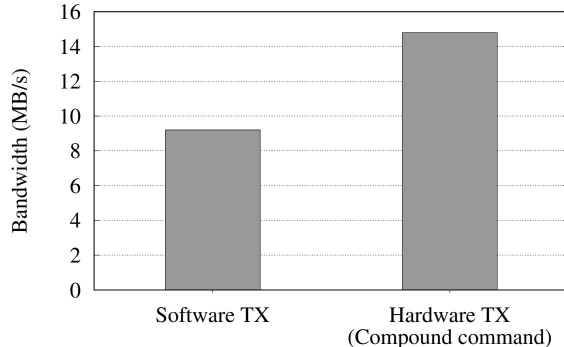

Figure 3: KVCeph bandwidth on different level of transaction support. Software TX indicates the bandwidth when the system lacks of hardware transaction support. Hardware TX indicates the performance when the hardware is aware of transactions with the compound commands.

To properly evaluate software-based transaction performance, we modified the KVCeph implementation to emulate write-ahead-logging (WAL). Entire key-value pairs for a transaction is written to a designated key (simulating the WAL writing) and the operation is followed by individual key-value operations. As shown in Figure 3, KVCeph gives 9.2 MB/s of storage bandwidth in this setting. If the hardware supports the transaction (i.e., using KVEMU with compound command support), we can omit the WAL writing and deliver the key-value operations using the compound command. In this case, KVCeph runs at 14.8 MB/s, which is 60.9 % of bandwidth improvement. We believe this result demonstrates the benefit of hardware transaction support enabled by compound commands.

4 Conclusion

We propose the compound command to amortize interfacing overhead between KVSSD and the host over NVMe. We made the case of the compound command by extending the current KVSSD command set, and demonstrated its benefit from a realistic workload. We are planning to implement the compound command in the real KVSSD and to evaluate its performance to verify the claims.

Acknowledgements

This work was supported by the National Research Foundation of Korea(NRF) grants funded by the Korea government(MSIT) (No. 2016R1A2A1A05005494 and No. 2018R1C1B5085902). Also, the ICT at Seoul National University provides research facilities for this study.

5 Discussion

Supporting complicated consistency As we discussed in Section 2.3, the current KVSSD can only provide limited consistency, but the limitation can be relaxed by adopting the techniques used by Amazon DynamoDB [1,23]. Specifically, DynamoDB APIs allow user applications to specify preconditions for updating operations, and DyanmoDB performs the operations only if all preconditions are met. For an example, when an application wants to atomically increase a value by 20, it can read the value, increases the original value by 20, and write back the result with a precondition specifying the original value. If the stored value and the precondition values are the same, the update can be safely applied, and otherwise,the transaction is aborted. By adopting the idea, we can define a transaction over split compound commands with preconditions appended. To realize the idea, the KVSSD command payload should be extended to express preconditions, and user APIs should be also revised accordingly. We are working on finding the best way to incorporate these features into the KVSSD command set and compound commands.

Mixed operation types in a transaction The current compound command can only merge the operations with the same type in a transaction. It was sufficient to run Ceph, and we also believe many applications only need this type of transaction. Nevertheless, this limitation can be relaxed by adding a field for each enclosed key-value pair specifying its operation type. In such a case, KVSSD should carefully check during the operation admission so that it does not make conflicts between operations in the same transaction.

Leveraging grouped key-value operations In this paper we focused on utilizing compound commands for supporting transactions, however, compound commands can be also leveraged for accelerate key-value access. Let assume that compound commands allow to define *a prefetch group* to notify KVSSD of the temporal locality of key-value pairs; when one of the grouped key-value pairs is accessed, KVSSD can prefetch other key-value pairs in advance expecting their access in a near future.

Quantitative analysis of memory copy and memory mapping Building compound commands in the packed command layout involves a number of memory copy and memory buffer allocation, and its overhead can outweigh the amortized interfacing overhead. However, compound commands are targeting for small key-value operations, and memory copy overhead will be small or comparable at the scale compared to building a scatter-gather list and to access each key-value pair via separate DMA. Nevertheless, we are working on quantitatively analyzing the overhead on real hardware.

Returning results of multiple operations When the compound command is used for simply collecting multiple operations, each operation might have different result. In this sense, KVSSD should have an interface to efficiently return the result of each operation. We currently assume that KVSSD generates a list of results and returns it to the host. And then the device driver on the host can parse the list and return each result accordingly.

References

[1] Amazon. Amazon DynamoDB transactions. https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/transactions.html.

[2] Amazon. Amazon S3. https://aws.amazon.com/s3.

[3] David G. Andersen, Jason Franklin, Michael Kaminsky, Amar Phanishayee, Lawrence Tan, and Vijay Vasudevan. FAWN: A fast array of wimpy nodes. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP)*, Big Sky, MT, October 2009. ACM SIGOPS.

[4] Apache Foundation. Apache Cassandra. https://cassandra.apache.org.

[5] Jeremy Condit, Edmund B. Nightingale, Christoper Frost, Engin Ipek, Benjamin Lee, Doug Burger, and Derrick Coetzee. Better i/o through byte-addressable, persistent memory. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP)*, Big Sky, MT, October 2009. ACM SIGOPS.

[6] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store, October 2007.

[7] Facebook. RocksDB transactions. https://github.com/facebook/rocksdb/wiki/Transactions.

[8] Brad Fitzpatrick. Distributed caching with memcached. *Linux Journal*, 2004, August 2004.

[9] Boncheol Gu, Andre S. Yoon, Duck-Ho Bae, Insoon Jo, Jinyoung Lee, Jonghyun Yoon, Jeong-Uk Kang, Moonsang Kwon, Chanho Yoon, Sangyeun Cho, Jaeheon Jeong, and Duckhyun Chang. Biscuit: A framework for near-data processing of big data workloads. In *Proceedings of the 43rd International Symposium on Computer Architecture*, pages 153–165, Seoul, South Korea, June 2016.

[10] Intel. Intel 3D NAND technology transforms the economics of storage. https://www.intel.com/content/www/us/en/products/docs/memory-storage/solid-state-drives/3d-nand-technology-animation.html.

[11] Intel. Intel Optane DC persistent memory. https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html.

[12] Intel. Intel Optane technology: Revolutionizing memory and storage. https://www.intel.com/content/www/us/en/architecture-and-technology/intel-optane-technology.html.

[13] JEDEC Solid State Technology Association. Embedded multi-media card (eMMC) electrical standard (4.5 device), June 2011.

[14] Hyeong-Jun Kim, Young-Sik Lee, and Jin-Soo Kim. NVMeDirect: A user-space i/o framework for application-specific optimization on NVMe SSDs. In *Proceedings of the 8th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage)*, Denver, CO, June 2016.

[15] Young-Sik Lee, Luis Cavazos Quero, Sang-Hoon Kim, and Jin-Soo Kim. Activesort: Efficient external sorting using active ssds in the mapreduce framework. *Future Generation Computing Systems*, December 2016.

[16] Huaicheng Li, Mingzhe Hao, Michael Hao Tong, Swaminathan Sundararaman, Matias Bjørling, and Haryadi S. Gunawi. The CASE of FEMU: Cheap, accurate, scalable and extensible flash emulator. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies (FAST)*, pages 83–90, Oakland, California, USA, February 2018.

[17] Inc. MongoDB. MongoDB: The most popular database for modern apps. https://www.mongodb.com.

[18] NVM Express, Inc. NVM Express. https://nvmexpress.org.

[19] Red Hat, Inc. Ceph. https://ceph.com.

[20] Samsung. Samsung V-NAND technology. https://www.samsung.com/us/business/oem-solutions/pdfs/V-NAND_technology_WP.pdf.

[21] Co Samsung Electronics. Samsung key value SSD enables high performance scaling.

[22] Woong Shin, Qichen Chen, Myoungwon Oh, Hyeonsang Eom, and Heon Y. Yeom. OS I/O path optimizations for flash solid-state drives". In *Proceedings of the 2014 USENIX Annual Technical Conference (ATC)*, Philadelphia, PA, June 2014.

[23] Doug Terry. Transactions and scalability in cloud databases—can't we have both? In *Proceedings of the 17th USENIX Conference on File and Storage Technologies (FAST)*, Boston, MA, February 2019. USENIX Association, USENIX Association.

[24] Unknown. Open memory platform development kit. http://github.com/OpenMPDK, 2018.

[25] USENIX Association. *A Blackbox Approach to Performance Modeling of KVSSDs*, Boston, MA, February 2019.

[26] Z. Yang, J. R. Harris, B. Walker, D. Verkamp, C. Liu, C. Chang, G. Cao, J. Stern, V. Verma, and L. E. Paul. SPDK: A development kit to build high performance storage applications. In *Proceedings of the 2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 154–161, December 2017.

[27] Sang yong Oh. KV SSD firmware introduction. https://github.com/OpenMPDK/KVSSD/wiki/presentation/kvssd_seminar_2018/kvssd_seminar_2018_fw_introduction.pdf.