KV-SSD: What Is It Good For?

Manoj P. Saha
School of Computing and Information Sciences
Florida International University
Miami, USA
msaha002@fiu.edu

Adnan Maruf
School of Computing and Information Sciences
Florida International University
Miami, USA
amaru009@fiu.edu

Bryan S. Kim
Department of Electrical Engineering & Computer Science
Syracuse University
Syracuse, USA
bkim01@syr.edu

Janki Bhimani
School of Computing and Information Sciences
Florida International University
Miami, USA
jbhimani@fiu.edu

2021 58th ACM/IEEE Design Automation Conference (DAC) | 978-1-6654-3274-0/21/$31.00 ©2021 IEEE | DOI: 10.1109/DAC18074.2021.9586111

***Abstract*—An increasing concern that curbs the widespread adoption of KV-SSD is whether or not offloading host-side operations to the storage device changes device behavior, negatively affecting various applications' overall performance. In this paper, we systematically measure, quantify, and understand the performance of KV-SSD by studying the impact of its distinct components such as indexing, data packing, and key handling on I/O concurrency, garbage collection, and space utilization. Our experiments and analysis uncover that KV-SSD's behavior differs from well-known idiosyncrasies of block-SSD. Proper understanding of its characteristics will enable us to achieve better performance for random, read-heavy, and highly concurrent workloads.**

***Index Terms*—storage, key-value database, solid state drive, KV-SSD**

I. Introduction

Embedded systems are elemental in powering connected automotive systems, smart homes, and internet of things (IoT) infrastructure. These systems often require flexible and straightforward data management techniques. Embedded key-value (KV) stores, such as RocksDB and LevelDB, cater to these needs by providing a straightforward interface for storing, searching, and filtering data. However, deploying KV stores in embedded systems on top of block storage leads to redundant data management overheads. Multiple layers of mapping have to be maintained to keep track of data conversions between variable-length KV pairs (KVPs) to files, then from files to fixed-size logical blocks, and finally from logical blocks to physical flash pages. These mapping and data conversion overheads give rise to CPU and memory contention in the resource-limited embedded system. Emerging Key-Value Solid State Drive (KV-SSD) technology [1], [2] promises to streamline these redundant data management overheads with direct data access, in-situ key-value data management, and better scaling.

However, KV-SSD is yet to meet mainstream adoption, despite its API ratification by SNIA (Storage Networking Industry Association) [3]. Unlike a block-SSD, whose performance behaviors and characteristics are better understood, those of a KV-SSD are still unfamiliar in the storage landscape, limiting its widespread use. Furthermore, although available publications [1], [4] describe the KV-SSD architecture and performance, the in-depth details of how the KV-SSD manages the complex interplay among data indexing, space allocation, and garbage collection is not explored. In the traditional I/O stack, on the other hand, the details of KV stores such as RocksDB and file systems such as ext4 are transparent at the source level to the users, making it easy to fine-tune the performance of the storage application. This is the first work to fill this knowledge gap in the KV stack by investigating answers to the following research questions (RQs) in detail:

RQ1: How does the performance of KV-SSD compare against that of block-SSD under a wide variety of workloads?
RQ2: What can we learn about the internal components and organization of KV-SSD through experiments, to use it more effectively in embedded systems?

In this work, we systematically measure, quantify, and characterize the performance of KV-SSD and block-SSD to understand the benefits and drawbacks of the two I/O stacks. To answer RQ1 we examine KV-SSD performance against two distinctly different KV stores deployed on block-SSD. To answer RQ2, we identify the major components of the new I/O stack and analyze their impact on I/O performance. For this study, we conduct thousands of hours of experiments on Samsung PM983 devices that can be configured either as KV-SSD or block-SSD. We analyze large amounts of performance data collected via various tools and share the most significant findings and observations.

We compare KV-SSD's performance with its block counterpart for CPU utilization, device bandwidth, and I/O latency under a wide array of workloads to answer RQ1. Our study yields that KV-SSD reduces CPU utilization by a factor of 13, on average, compared to RocksDB on block storage. However, KV-SSD shows as low as 0.44× and 0.22× bandwidth utilization than block-SSD direct I/O for 4KB random reads and writes, respectively. The latency of direct I/O operations on KV-SSD can be as high as 2.63× for writes and 8.1× for reads, compared to block-SSD. When compared to end-to-end latency of operations on RocksDB (with ext4 file system and 10MB block cache) or Aerospike (with direct I/O), KV-SSD provides up to 23.08× and 3.64× better performance for inserts and updates, respectively.

Our experiments also uncover three internal components that dominate the I/O performance of KV-SSD—the hash-based indexing scheme, KV-specific key handling, and data packing policies, and host-side KV command set. First, the hash order-based index operations have upended benefits of sequential access on block-SSD—both in terms of I/O latency and index size—causing up to 16.4× latency hikes as the index size increases. The key handling and data packing policies of KV-SSD also play a major role behind this latency increase, although it helps in reducing read and write latency as much as 0.37× and 0.86×, respectively, for KVPs smaller than

This work was partially supported by the National Science Foundation (NSF) Awards CNS-2008324 and CNS-2122987, and by a KV-SSD equipment grant from Samsung.

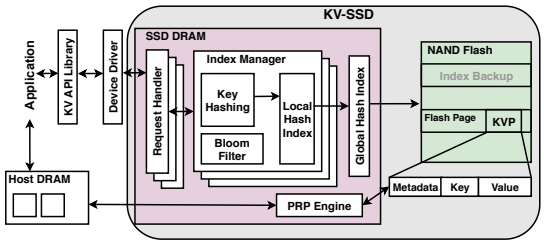


Fig. 1: KV-SSD I/O path.

24KB at a high request scale (queue depth 64). However, the internal organization of data in KV-SSD limits the maximum number of KVPs that can be stored (∼3.1 billion on a 3.84TB Samsung PM983 KV-SSD) and can induce up to 20× space amplification. Finally, the inefficiency in host-side KV-aware NVMe command processing can also reduce bandwidth utilization of KV-SSD by a factor of 0.53× for larger keys. We make relevant experimental data publicly available at https://support.cis.fiu.edu/ftp/damrl/ for the research community to understand and model the performance behavior of KV-SSD.

II. Background

The Key-Value SSD from Samsung [1] is a NAND flash storage device with a KV Flash Translation Layer (FTL), adapted from block FTL. The NVMe KV-SSD supports the storage, retrieval, and deletion of variable-length keys and values (with key-length limited from 4B to 255B keys and value-length limited from 0B to 2MB, currently). The SSD supports internal operations on variable-length keys by hashing them into fixed-length key hashes. The variable-length KV pairs, along with additional metadata, are stored as variable-length blobs in a log-structured manner inside KV-SSD. The physical locations of these blobs are stored in the KV-FTL that uses a multi-level hash table structure as the main index [1], [2].

User applications can request KV operations using the SNIA KVS API library, which communicates with the device driver that ultimately talks to KV-SSD. The SNIA KVS API library supports fundamental KV operations, including store, retrieve, delete and exist, both in synchronous and asynchronous mode. The API can access the storage device through either a kernel device driver (KDD) and a SPDK-based user-space driver (UDD) [2].

Figure 1 shows the I/O path of data in KV-SSDs. First, a user application requests an operation through the KV API. The KV API then talks to the device driver, and the device driver submits the request in the device I/O queue using an appropriate vendor-specific NVMe command for the KV interface. A request handler then initiates a data transfer operation using Physical Region Page (PRP) between the host system and SSD DRAM, based on the command. During store operations, the key is hashed by one of the index managers[1] and temporarily stored in a local index before merging it with the global index of the KV-SSD. Once the indexing information for a key is merged in the global/main index, the KVP is programmed on NAND flash pages. Each KVP consists of metadata, key, and value. The metadata stores information such as the key size, value size, and namespace. In addition to the global index, the key is also stored in an iterator bucket for iterator management, based on the first 4 bytes of the key. On the other hand, read requests need to go through membership checking to ensure that the right data is being returned. Index manager-resident Bloom filters can be leveraged to quickly resolve read or exist queries for non-existent keys.

In contrast, block-SSD supports the block-level interface of storage access protocols such as SCSI, SATA, and NVMe. In the block-based storage systems, data is mapped to fixed-size logical blocks on the host side. These logical blocks are mapped to fixed-size NAND flash pages (or blocks) in the SSD. Typically, the file system maintains a mapping of files to logical blocks, and the SSD maintains a logical blocks-to-physical mapping, known as the Flash Translation Layer (FTL). Hence, the logical granularity of block-SSD still remains to be fixed-size logical blocks on the host side, whereas the physical granularity of access inside the SSD for store/retrieval can be a flash page, usually of sizes between 4KB-32KB. However, to use page-level physical access granularity, the FTL needs to maintain a logical block to physical page mapping. Each physical SSD block contains around 4K pages, so maintaining a logical block to physical page mapping in FTL will consume a large amount of SSD DRAM. Thus, typically within a block-SSD, FTL maintains a logical block to physical block mapping that restricts accessing the storage device only using a fixed block size.

[1]Multiple index managers are used to reduce contention of the global index structure.

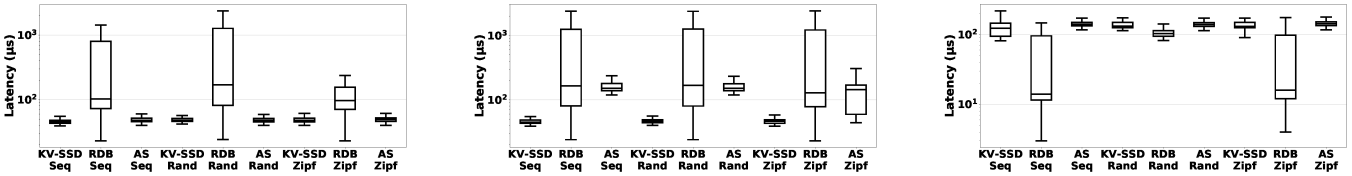
III. Methodology

Existing works related to KV-SSD focuses primarily on studying performance from a system utilization and scalability perspective, largely overlooking the individual impact of the newly introduced components of KV-FTL. In this study, we systemically analyze the impact of each major component of the newly proposed I/O stack and measure the end-to-end performance of KV-SSD against block-SSD-based KV stores. We identify three fundamental changes in the new I/O stack compared to block I/O. First, a KV-SSD accepts variable-length keys requiring it to implement a multi-level hash index structure. Second, the variable-length values also present a stark difference from the prevalent fixed-length logical blocks. In the traditional I/O stack, the file system is responsible for mapping a file (of variable-length name and variable-size) to a set of fixed-length logical blocks. Lastly, existing host-side optimizations are no longer effective, setting new rules for improving performance. We design our study to understand the ramifications of these changes on device I/O performance.

We established the means of our study in accordance with the precedence set by [5] to answer the following questions.

- Which benchmarks should we use to analyze KV-SSD behavior?
- Which KV stores should we study?
- What kind of access patterns should we study?
- How do we identify the root cause of KV-SSD behavior?

Since the block I/O and KV I/O stacks are so different, it is important to ensure a fair comparison between the two, as not to introduce any bias in the study. For a fair evaluation of both KV-SSD and block-SSD, we use OpenMPDK KVBench [2], an open-source benchmark based on ForestDB's benchmark [6] with a configurable workload generator. KVBench is comparable to db_bench, a built-in microbenchmark for LevelDB and RocksDB. It can generate KV workloads representative of real-world scenarios. KVBench generates a series of KV operations with variable-length keys and values, and the access patterns can be configured to be sequential, uniformly random, or skewed following a Zipfian distribution. These I/O requests are submitted through the KV interface using the Kernel Device Driver (KDD) or the SPDK-based User Device Driver (UDD). An alternative to KVBench would be YCSB. Although it is a popular real-world benchmark for KV storage, it requires a database engine in the middle that properly interfaces with the KV-SSD. Such modified database engines are not available at our disposal at this moment and would take considerable time and effort to develop. Thus it is out of the scope of this study.



(a) Insert **(b)** Update **(c)** Retrieve

Fig. 2: The hash-ordered index operations in KV-SSD take away the benefits of sequential access. In the figures, Seq, Rand, and Zipf stand for Sequential, Random, and Zipfian, respectively. Above, we compare KV-SSD, RocksDB (RDB), Aerospike (AS) I/O latency. Insert(a) and read(c) operations show marginally better latency for sequential access over Uniform random and Zipfian workloads. However, all workload distributions provide similar latency for update operations. KV-SSD also performs better than RocksDB for inserts and updates, whereas it only provides better performance for updates in Aerospike.

We approach RQ1 by comparing the performance of the KV-SSD accessed through KDD against that of (1) RocksDB on an ext4 file system and a block-SSD, and (2) Aerospike with direct access to the block-SSD. The LSM-tree-based data management approach of RocksDB is representative of the majority of popular KV stores at this moment. Whereas the hash-index-based Aerospike closely matches KV-SSD's own internal metadata management technique. We examined these distinct systems using a wide variety of workloads using KVBench, varying the following workload parameters: (1) access pattern (sequential, uniform, Zipf), (2) request type (insert-only, update-only, read-only, or mixed), and (3) request size (key size, value size, or block size). The KVPs are accessed asynchronously without data caching unless otherwise noted. Furthermore, we use the same SSD hardware (Samsung PM983) that can be configured as either a KV-SSD or a block-SSD, depending on the firmware; this eliminates any possibility of performance difference due to the SSD's hardware architecture.

To answer RQ2, we examine the intricacies specific to KV-SSD by analyzing data from multiple sources: KVBench logs, dstat, iostate, S.M.A.R.T, and NVMe-CLI. These data, in conjunction with our understanding of the KV API and the KV-SSD literature, are used to identify potential performance bottlenecks and their root causes. Although the KV-SSD is effectively a black-box with its internal implementation details unknown, we present our understanding of some of the internal behavior of the KV-SSD through careful analysis. We run our experiments using KVBench and custom scripts that use either the KV API or IOCTL for direct access.

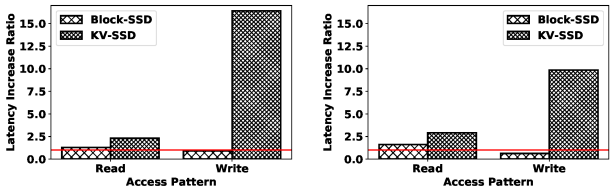
We run our experiments on two systems with the same configurations: 2x Intel Xeon Silver 4208 CPU @ 2.10GHz processors, 192GB DDR4 DRAM (which was reconfigured to 6GB for certain macro-level experiments), and two Samsung PM983 NVMe KV-SSDs (firmware version ETA51KCA for KV-SSD and EDA53W0Q for block-SSD). Both systems run the Ubuntu operating systems, and we use KDD for all our experiments.

IV. Results and Analysis

In this section, we present our results and analysis to measure, quantify, and understand the performance of KV-SSD. We begin by comparing the end-to-end performance of KV-SSD and block-SSD. Then to better understand the performance of KV-SSDs, we study the impacts of offloading the index structure and data packing to the storage device. Finally, we discuss the influence of the host-side KV software stack on performance.

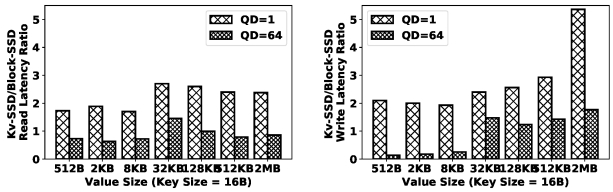
Performance impact of moving key-value management operations from host to NVMe storage: The primary motivation behind the design of KV-SSD is to simplify the I/O stack and reduce I/O handling related host resource utilization, by offloading page translation, index management, and iterator management tasks of key-value applications, to the storage device [1], [4]. Surprisingly, KV-SSD needs only standard SSD hardware, which is augmented by a new Flash Translation Layer (FTL) firmware that provides its processing capabilities to support direct key-value accesses [1]. Intuitively, the additional key-value processing on existing hardware increases the I/O latency of KV-SSD. The average retrieve and insert latency of KV-SSD for random workloads is 1.7× and 2.5× higher than block-SSD direct I/O [7]. However, we cannot decide the merits and demerits of the new storage stack using KV-SSD without comparing its end-to-end performance with traditional KV stores using block-SSD. Therefore, in Fig. 2, we compare end-to-end I/O performance of KV operations on KV-SSDs, and software KV stores such as RocksDB and Aerospike deployed on top of block-SSD. We issue 10 million I/O requests (insert, update or retrieve) of 16B keys and 4KB values, with different I/O access patterns such as sequential, uniform random, and Zipfian distributions. We found that the host CPU utilization reduced up to 0.92× while using KV-SSD compared to RocksDB deployed on top of block-SSD. CPU utilization reduction compared to Aerospike was much less, since it does not require complex compaction and lookup operations of LSM-tree-based RocksDB. *Interestingly, further analysis revealed that while host-side CPU utilization was reduced by using KV-SSD, end-to-end average I/O latency is significantly impacted depending upon the workloads.* The I/O latency distribution of Fig. 2 shows, KV-SSD performs better than RocksDB (with only 10MB block cache) for inserts (Fig. 2a) and updates (Fig. 2b), but suffers significantly during retrieves (Fig. 2c). While compared to Aerospike, KV-SSD performs better only for updates (Fig. 2b). We anticipate that the increase in the KV retrieve latency in KV-SSD, compared to block-SSD, might be due to the indexing, data packing, and key-handling operations moved from the host to the storage device. However, the lack of proprietary knowledge of KV-FTL and flash controller workings limit our ability to measure exact resource utilization of the CPU and memory within KV-SSD. Thus, we further analyze the impact of indexing, data packing, and key-handling operations by performing direct access to KV-SSD from the host.

Impact of key-value indexing: As explained earlier in Sec. II, in KV-SSD, the block-SSD FTL is extended to support variable-size key-value pairs using a multi-level hash table for fast point query as a global index structure. In KV-SSD, to enable easy and efficient management of the index structure, variable-length keys are transformed into fixed-length key hashes. Then these key hashes are used for physical page translations. This hashing mechanism makes key handling and index management easy. However, the sequential access pattern of workloads may no longer imply sequential access to the storage device due to hashing. While we popularly know that sequential reads or writes to flash results in much better performance compared to random I/O [5]. For example, the datasheet of Samsung PM983 NVMe block-SSD, as well as our experiments, reveals that sequential reads and writes incur up to 0.8× and 0.6× lower latency



(a) Average latency (b) Tail latency (99th percentile)

Fig. 3: KV-SSD's multi-level hash index can become too large to manage efficiently as the number of KVPs stored in the device increases, impacting device performance drastically. Read and write latency of KV-SSD can increase up to 2× and 16.4× as the number of keys reaches close to the maximum limit (from 1.5 million to 3 billion KVPs), compared to block-SSD's near-constant performance. The horizontal line in the figures represents no change in I/O latency.



(a) Read latency ratio (b) Write latency ratio

Fig. 4: KV-SSD I/O performance suffers due to its log-like data packing policy and key handling overheads. However, its byte-aligned log-like data packing policy can provide better performance for KVPs smaller than a flash page at high concurrency, compared to block devices. (a) and (b) shows average latency ratios of KV-SSD vs. block-SSD for read and write operations, respectively (<1 is better for KV-SSD).

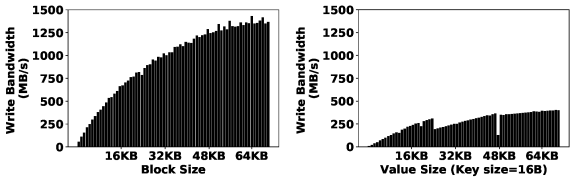
than their random counterparts while performing 4KB I/Os using block-based firmware [7]. It cannot be guaranteed whether the same benefit of sequential I/Os still holds for the PM983 with key-value firmware. We study the latency of insert, update and read operations on KV-SSDs. *We observe that hash-based design in KV-SSD takes away the advantage of better performing sequential accesses.* Fig. 2 shows that the performance for random and sequential workloads on KV-SSD is almost the same. Sequential workloads in block-SSD FTL minimize metadata management and lookup by storing smaller amounts of metadata and optimize data storage by intelligently exploiting the parallelism of the flash device [8]. Since key hashes change the order of the keys, the multi-level hash table index in KV-SSD does not store keys in sequential order. This increases metadata storage and lookup overheads. In addition, hash-order-based indexing also nullifies the use of the same sequential data storage optimizations of block-SSD in its KV counterpart. Hence, sequential workloads in KV-SSD no longer provide the same performance benefits as in block-SSD.

Impact of index occupancy: The global hash-based index in KV-SSD needs to maintain a separate record for each key. This increases the size of the index almost linearly with the number of KVPs inserted in the storage device. Quick metadata insertion and lookup in the index can be ensured by keeping it entirely in SSD DRAM. However, an increase in the size of the index, upon insertion of more keys, may make the index structure too large to fit in SSD DRAM, directing index operations to flash. *We observe that upon overflow of the index structure from the SSD DRAM to flash pages, the latency of the read and write operations is tremendously impacted.* If an entry is not found in the index cache, then it would invoke a series of flash page reads to access the desired index entry from a large multi-level index structure. Fig. 3 shows the average latency of read and write operations for KV-SSD at low and high index occupancy compared to the performance of the block-SSD with the same amount of the prior occupied storage capacity. Particularly, to test performance at low occupancy, we write 1.53 million KVPs with 16B keys and 512B values on KV-SSD, and for high space usage, we fill up the SSDs with 3 billion KVPs of the same size. *Surprisingly, we see that the average latency of the KV-SSD increases up to 2× for reads and even more 16.4× for writes with the increased index occupancy.* While conducting similar experiments on the block-SSD, by filling the same amount of 512B blocks as that of KV-SSD (i.e., 1.53 million 512B blocks for low occupancy and 3 billion 512B blocks for high occupancy), we see the performance almost remains same. This reinforces that the degradation in the KV-SSDs performance is surely not due to the NAND flash cells. We anticipate that the performance of the block-SSD does not change drastically, since hybrid-FTLs in block-SSD operate on a fixed number of logical blocks and can optimize the size of the index by minimizing the number of entries per logical block [9]. In contrast, the total number of KVPs that need to be stored on KV-SSD depends on the workload, and offset information for each pair needs to be maintained separately.

Impact of additional data packing and key handling operations within KV-SSD: In block-based I/O stack, variable-length data to fixed-length block conversion is handled by the host, which then flushes the fixed-length blocks to the SSD to be programmed on fixed-length physical pages or blocks. In KV stack, the job of packing variable-length data into fixed-length physical pages is offloaded to the storage device that performs it in a log-like manner [1]. So, KV-SSD has to employ additional data packing operations other than key indexing to enable direct I/O operations for variable-length KVPs. To better understand such internal data packing operations, we study the performance with respect to various value sizes and concurrency. Further, we analyze the effect of data packing on garbage collection and space utilization.

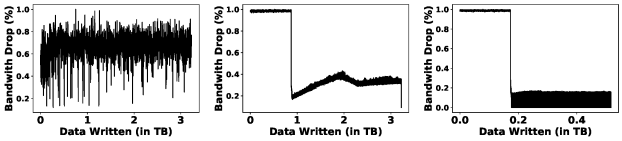
Previous research [10] attributes the higher bare metal access latency of KV-SSD to the key handling overheads of the device. Since additional tasks such as key hashing, membership checking, and merging of the local and global index have to be executed for each key inside the storage device [1]. Hence, key handling overheads increase the time required to complete each I/O request in KV-SSD. *However, to our surprise, we notice that the key handling overheads are not the only reason behind the higher I/O latency of KV-SSD. The internal data packing activities significantly impact the KV-SSD performance.* In Fig. 4, we analyze the direct access latency ratio of KV-SSD and block-SSD for read and write operations while increasing the value size and concurrency (i.e., queue depth). We perform in total the same 1.53 million KV or block I/Os for each value size in Fig. 4. Thus, hypothetically if the increase in latency was only due to the key handling overhead, then for the same number of KVPs, the additional performance overhead should remain the same for different value sizes. But Fig. 4 contradicts the above hypothesis, implying that in addition to key handling overhead also some other factor(s) accounts for KV-SSD's performance.

Impact of concurrency: *Our experiments reveal that KV-SSD can pack (during store) and unpack (during retrieve) variable length-data more efficiently, compared to a block-SSD, when there are a large number of concurrent I/O requests, especially with relatively small I/O sizes.* In Fig. 4, we observe that the latency of KV-SSD can be up to 5.4× higher than block-SSD, but with higher concurrency (i.e., queue depth = 64) the KV-SSD performs better. Retrieve/read operations consumes lower or equal time to complete than block-SSD, while for store/write operations, KV-SSD shows 0.86× latency



(a) Block-SSD (b) KV-SSD

Fig. 5: KV-SSD's key handling and data packing overheads strangle device bandwidth utilization. Its byte-aligned log-like data packing creates further overheads as the KVP size gets bigger than a flash page size, triggering data splitting and offset management operations. Hence, we observe intermittent drops in device bandwidth, as shown in (b).



(a) RocksDB random update (b) KV-SSD random update (c) KV-SSD pseudo-random update

Fig. 6: Byte-aligned log-like data packing in KV-SSD makes it susceptible to foreground GC operations. (b) and (c) show bandwidth drop due to foreground GC during random updates, while (a) shows no GC triggered during random updates in RocksDB on block-SSD. (c) stopped before all update completion due to time restrictions.

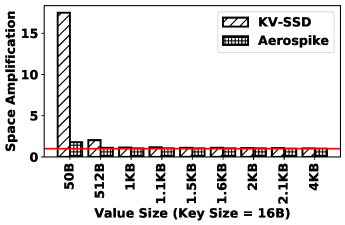


Fig. 7: KV-SSD can suffer from high space amplification (i.e., actual SSD space utilization/data written by application) due to internal padding added to small KVPs. Hash index-based Aerospike on raw block-SSD shows space amplification of less than 2. LSM tree-based RocksDB's space amplification is 1.111... in the worst case, without considering files system metadata overheads [12]. KV-SSD's high space amplification limits the maximum number of KVPs that can be stored in the device.

reduction than block-SSD for smaller value sizes (see Fig. 4a). We believe this is because the FTL within KV-SSD better allows it to take advantage of the internal SSD parallelism among multiple channels, dies, and planes for concurrent I/O requests, as the data packing for small KVPs within KV-SSD is simple compared to block-SSD. We elaborate on the above in detail soon. These performance statistics clearly suggest that the data packing policy of KV-SSD works as a boon for performance for values smaller than 32KB at high concurrency and as a bane for performance for values equal or larger than 32KB.

Impact of value size: As we see from Fig. 4, *the data packing policies of KV-SSD favor relatively smaller value sizes, especially for store operations.* KV-SSD stores metadata, key, and value of each KVP as variable-length blobs within flash pages (usually between 4KB and 32KB in size), in a log-like manner [1] without performing any rearrangements to KVPs in the buffer before programming them to flash pages. In contrast, even if block-SSD FTLs write data in a log-like manner, they have an incentive to maintain the sequentiality of blocks in the physical space [8]. Storing logical blocks sequentially in physical space reduces index size. Hence, we assume, block-SSD FTL tries to reorganize data and/or hold data in buffer much longer before flash programming (writes) happens. But, KV-SSD does not have any incentive in maintaining the sequentiality of keys during data packing due to its hash-based index structure. Thus, for small KVPs (e.g., 512B, 2KB, and 8KB) that fit within a single page along with its metadata, the write latency of KV-SSD is up to 0.86× smaller than the block-SSD, as data packing within KV-SSD has no additional rearrangement operations. We believe similar benefits are not observed at queue depth of one since it takes much longer to fill a flash page buffer at a low request scale, and key handling overheads dominate the I/O latency. However, if a KVP can't be packed within a single flash page, then data packing within KV-SSD needs to perform the additional work of splitting and packing the data into multiple flash pages [11], along with additional offset pointer management that drastically increases KV-SSDs write latency up to 5.4× compared to block-SSD.

We further validate our above-mentioned hypothesis about variable-length blob-based data packing in a log-like manner within KV-SSD by observing the write bandwidth of KV-SSD upon increasing the value size. Fig. 5 shows, unlike block-SSD, the bandwidth of KV-SSD drops significantly for some value sizes such 25KB, 49KB, etc. We anticipate this is because the physical page size of our KV-SSD is probably 32KB, which can fit up to 24KB of value size in addition to metadata and key. Moreover, each physical page may also have some space reserved for data recovery operations, such as erasure coding. Thus, if our hypothesis is valid, upon the increase in KVP size, when it no longer fits within a single page, then due to the additional splitting, packing, and offset pointer management operations, KV-SSD bandwidth should drop (see Fig. 5b). Similar zig-zag bandwidth changes are not seen in block-SSD (see Fig. 5a), because its FTL programs logical blocks to the flash pages without modification.

Problem of foreground garbage collection (GC) within KV-SSD: *Interestingly, we observe that the data packing strategy that enables better performance for smaller KVPs at high concurrency also makes KV-SSD more susceptible to foreground GC operations, compared to block-SSD.* Foreground GC occurs when many write/update requests arrive, and there is no more physical space left in the SSD. The bandwidth drops due to foreground GC, as write/update requests need to wait until GC can free up some space. To observe the impact of foreground GC on the device bandwidth, we first fill 80% of SSD device capacity with 16B keys and 4KB values, then run uniform-random or pseudo-random[2] update workloads for all stored keys, rewriting the same amount of data. We observe that just a small amount of randomness in updates can trigger foreground GC in KV-SSD. However, triggering foreground GC for 4KB values in block-SSD is extremely tough both for direct I/O and file systems. In Fig. 6a, no drastic performance drop is observed. The level-based data arrangements in RocksDB lead to sequential access patterns of files on block-SSD that erase entire blocks for invalid SST files, reducing chances of triggering GC. *Hence, in KV-SSD, bursty workloads may intensively suffer from low performance if the drive capacity is almost filled.*

Problem of space amplification within KV-SSD: *Although KV-SSD provides better latency for small KVPs, it adds up to 1KB padding to small KVPs, incurring space amplification of up to 20×.* Figure 7 shows that KVPs with 50B values incur space amplification of 17× in KV-SSD, whereas it is only 1.8× for Aerospike on raw

[2]We move a small sliding window across the whole distribution of keys from the insert phase, and randomly choose keys from within the sliding window.

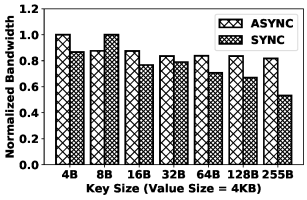


Fig. 8: Samsung's KV command set design for NVMe interface penalizes I/O performance for large keys. Keys larger than 16B needs two NVMe commands to pass the key to the KV-SSD, increasing processing overheads.

block-SSD. In addition, this also limits the number of KVPs that can be stored in KV-SSD. For example, we observed that the maximum number of KVPs that can be stored in a 3.84TB Samsung KV-SSD is approximately 3.1 billion. While the exact reason behind such a limit is hard to know due to many proprietary and unknown details about KV-SSD, we explain our anticipations below. Our first assumption is that the internal access granularity of KV-SSD is 1KB, twice the size of two 512B sectors of the block-SSD. In that case, KVPs which are larger than 1KB but smaller than 2KB should be padded up to 2KB. However, from Figure 7, we see that KV-SSD packs data very tightly beyond 1KB, achieving close to 1 space amplification for KVP sizes ranging from 1KB to 4KB. Our second assumption is the Error Correction Code (ECC) requirements have influenced 1KB padding of small KVPs. ECC modules are integrated with flash controllers and cannot be modified in software. The size of ECC sectors, the minimum data unit for ECC computation, are also predefined in hardware. Hence, KV-SSD has inherited the ECC requirements of the block-SSD hardware, given the former is implemented only in firmware. However, since KV-SSD stores metadata and keys along with data values, it has a higher reliability requirement than block-SSD. If the ECC sector of the device is 1KB, then packing multiple KVPs within this space will increase the risks of data corruption for multiple KVPs. Since data is packed as variable-length blobs in flash pages, this may have cascading effects. Hence, it is a possibility that to reduce data reliability issues, KVPs are internally limited to a 1KB minimum size. Lastly, we assume that this may be due to constraints of managing an index with good performance and hash collision resolution. Since the range of keys available to KV applications is practically unlimited, KV-SSD runs the risk of ending up with an unfathomably large index structure with lots of small KVPs stored in the device. Given the limited resources of a flash controller, maintaining reasonable latency of operations and low hash collisions in the hash-based global index structure is extremely hard.

Impact of new host-side software stack: Finally, we study how host-side design decisions impact I/O performance to identify three major inefficiencies in the current vendor-specific commands to enable KV operations over NVMe. *First, as keys are passed to KV-SSD (from the host) inside NVMe commands, one KV operation might require more than one NVMe command to be issued [13].* Each KV API request is passed to the SSD as a 64B NVMe command, and each command has 16B reserved space for a key. If the key size is larger than 16B, it requires an additional NVMe command to pass the key to the storage device. Fig. 8 shows how operations with key length larger than 16B lowers device bandwidth utilization, both for synchronous and asynchronous I/O. *Second, for extremely small KVPs, unnecessary overhead is incurred due to the fixed size of each NVMe command (64B).* For example, as observed in [14], the average KVP size in Facebook's real KV store deployments is between 57B and 154B. Issuing one or more 64B NVMe commands to store or retrieve KVPs, which are only a hundred bytes or less, is a waste of critical system resources. Hence, [10] proposed consolidation of multiple NVMe commands into one compound command for smaller KVPs to increase device performance. *Third, unlike its block counterpart, the NVMe write commands for KV-SSD do not send critical metadata information such as expected access frequency or acceptable access latency to the device.* Such information may help in designing efficient data-placement strategies for KV-SSDs based on the hotness or coldness of data.

V. Conclusions

This paper poses two research questions for investigation – How does KV-SSD perform in comparison to block-SSD? How does KV-SSD work under the hood in the context of embedded systems? KV-SSD is a good fit for embedded systems, as they enable various operations like KV insert and update at lower I/O latency and also provide better performance at high concurrency. In addition, using KV-SSD can reduce the burden on the small CPUs of the IoT devices. However, it is better to avoid KV-SSD for write-heavy workloads, especially with extremely low data size, due to its susceptibility to foreground GC and high space amplification for small values. In future, we plan to explore KV-SSD performance behavior under real-world workloads and benchmarks, such as YCSB. We also plan to develop an analytical model of KV-SSD performance that can help researchers generate more representative workloads compatible with KVBench.

References

[1] Y. Kang, R. Pitchumani, P. Mishra, Y.-S. Kee, F. Londono, S. Oh, J. Lee, and D. D. Lee, "Towards building a high-performance, scale-in key-value storage system," in *Proceedings of the ACM International Conference on Systems and Storage (SYSTOR)*, 2019, pp. 144–154.

[2] Samsung. (2018) OpenMPDK KVSSD. [Online]. Available: https://github.com/OpenMPDK/KVSSD/

[3] SNIA. (2020) Key value storage API specification. [Online]. Available: https://www.snia.org/keyvalue

[4] R. Pitchumani and Y.-S. Kee, "Hybrid data reliability for emerging key-value storage devices," in *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, 2020, pp. 309–322.

[5] J. He, S. Kannan, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "The unwritten contract of solid state drives," in *Proceedings of the European Conference on Computer Systems (EuroSys)*, 2017, p. 127–144.

[6] Couchbase. (2020) ForestDB-Benchmark. [Online]. Available: https://github.com/couchbaselabs/ForestDB-Benchmark

[7] Lenovo. (2020) Lenovo PM983 Entry NVMe PCIe SSDs. [Online]. Available: https://lenovopress.com/lp0879.pdf

[8] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. Manasse, and R. Panigrahy, "Design tradeoffs for SSD performance," in *Proccedings of the USENIX Annual Technical Conference (ATC)*, 2008, p. 57–70.

[9] D. Ma, J. Feng, and G. Li, "A survey of address translation technologies for flash memories," *ACM Computing Surveys*, vol. 46, no. 3, 2014.

[10] S.-H. Kim, J. Kim, K. Jeong, and J.-S. Kim, "Transaction support using compound commands in key-value SSDs," in *Proceedings of the USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage)*, 2019.

[11] Y.-T. Chen, M.-C. Yang, Y.-H. Chang, T.-Y. Chen, H.-W. Wei, and W.-K. Shih, "KVFTL: Optimization of storage space utilization for key-value-specific flash storage devices," in *Proceedings of the Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2017, pp. 584–590.

[12] S. Dong, M. Callaghan, L. Galanis, D. Borthakur, T. Savor, and M. Strum, "Optimizing space amplification in RocksDB," in *Proceedings of the Conference on Innovative Data Systems Research (CIDR)*, 2017.

[13] Samsung. (2018) KV-SSD Seminar. [Online]. Available: https://github.com/OpenMPDK/KVSSD/wiki/presentation/kvssd_seminar_2018/kvssd_seminar_2018_fw_introduction.pdf

[14] Z. Cao, S. Dong, S. Vemuri, and D. H. Du, "Characterizing, modeling, and benchmarking RocksDB key-value workloads at Facebook," in *Proceedings of t he USENIX Conference on File and Storage Technologies (FAST)*, 2020, pp. 209–223.