Towards Building a High-Performance, Scale-In Key-Value Storage System

Yangwook Kang, Rekha Pitchumani, Pratik Mishra, Yang-suk Kee, Francisco Londono, Sangyoon Oh, Jongyeol Lee, *and* Daniel D. G. Lee.
Samsung Semiconductors, Inc.
(yangwook.k,r.pitchumani,mishra.p,yangseok.ki,f.londono,sangyoon.oh,jyjohn.lee,daniel00.lee)@samsung.com

Abstract

Key-value stores are widely used as storage backends, due to their simple, yet flexible interface for cache, storage, file system, and database systems. However, when used with high performance NVMe devices, their high compute requirements for data management often leave the device bandwidth under-utilized. This leads to a performance mismatch of what the device is capable of delivering and what it actually delivers, and the gains derived from high speed NVMe devices is nullified. In this paper, we introduce KV-SSD (Key-Value SSD) as a key technology in a holistic approach to overcome such performance imbalance. KV-SSD provides better scalability and performance by simplifying the software storage stack and consolidating redundancy, thereby lowering the overall CPU usage and releasing the memory to user applications. We evaluate the performance and scalability of KV-SSDs over state-of-the-art software alternatives built for traditional block SSDs. Our results show that, unlike traditional key-value systems, the overall performance of KV-SSD scales linearly, and delivers 1.6 to 57x gains depending on the workload characteristics.

CCS Concepts

• **Computer systems organization**; • **Hardware → Emerging architectures**; **Communication hardware, interfaces and storage**;

Keywords

Key-value store, Scalability, Key-value SSD

SYSTOR'19, June 2019, Haifa, Israel

ACM ISBN 978-1-4503-6749-3…$15.00
https://doi.org/10.1145/3319647.3325831

ACM Reference Format:
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 ACM International Systems and Storage Conference (SYSTOR'19).* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3319647.3325831

1 Introduction

With exponential growth of data, modern web-scale data-dependent services and applications need higher capacity and performance from their storage systems [8]. Such applications need to manage large number of objects in real time, but traditional object storage systems such as key-value stores are inefficient in handling the workloads in a scalable manner utilizing the full bandwidth of SSDs [20, 21].

One of the main obstacles for key-value stores is the IO and data management overheads to generate high-bandwidth IOs for SSDs. For example, RocksDB is designed to utilize large buffered IOs with small foreground processing using a log-structured merging mechanism [2]. However, it requires a background cleanup process called *compaction*, which reads the written data back and merge them to reorganize valid key-value pairs. Therefore, the amount of IOs and computations increases and often suffers from a slow-down or a stall when the background operation progresses slowly. Similarly, the defragmenter of Aerospike also requires CPU-intensive merge operations to manage updated key-value pairs [1].

Another obstacle is the overhead of maintaining the consistency of key-value pairs. Owing to the semantic gap between block and key-value requests, storage systems typically maintain their own metadata and use write-ahead logging technique (WAL) to prevent the metadata of key-value pairs from being written without data, or vice versa. This process creates dependencies between IOs and increases write amplification, slowing down the request processing. It also internally goes through two to three translation layers in the storage stack. Each incoming request is first processed and added into WAL, data, and metadata files. Then a file system re-indexes them by creating its own metadata such as inodes and journals for the files. Finally, the Flash Translation Layer (FTL) in an

SSD performs an additional translation from logical block addresses (LBA) to physical page numbers (PBN).

It has been challenging to reduce such overheads with less CPU resources and smaller write amplification for high scalability and performance. These issues get more pronounced for fast storage devices due to shorter time budget for request processing. For example, to saturate a modern NVMe SSD that can process 4 KB requests at the rate of 600 MB/s, a system needs to generate 150K 4 KB requests per second to the device, which is less than 7 µs per request. If the average request processing time of system increases to 30 µs, at least 4 CPUs are required to generate the same rate of IOs to the device. Therefore, as the number of devices increases, the high CPU demand to saturates the devices intensifies resource contention in a host system, limiting in-node scalability.

To alleviate these issues, we introduce KV-SSD that internally manages variable-length key-value pairs, providing an interface that is similar to that of conventional host-side key-value stores. By offloading the key-value management layer to an SSD, the need for host system resources can be significantly reduced and the overhead of maintaining key-value pairs can be isolated within a device. Moreover, to our best knowledge *KV-SSD* is the first Key-Value SSD prototype that supports large keys and unique functionalities such as *iterator*. Therefore, applications can directly communicate with *KV-SSDs* without going through additional indirection layers for name and collision resolution.

We design and develop KV-SSD on Samsung's datacenter-grade NVMe SSD hardware to study the performance implication of the in-storage key-value management in a realistic environment. We explore the potential benefits of KV-SSD to improve performance, resource-efficiency, and scalability for key-value store applications, evaluating it against state-of-the art conventional key-value stores, such as RocksDB and Aerospike. We show that, unlike conventional key-value stores, the aggregated performance of KV-SSDs scales linearly with the number of devices and fully utilize the devices with limited host-side compute and memory resources. Our evaluation shows that when using 18 NVMe devices per NUMA node, KV-SSDs outperforms conventional key-value stores by 1.6 to 57 times.

2 Motivation and Background

In this section, we describe the resource requirements of various components in the I/O stack and their impact on utilizing low latency, and high-bandwidth SSDs. We first investigate on the minimum resources required to saturate NVMe SSDs, and present the detailed analysis on the repercussions that the architectural designs of conventional key-value stores have on the amount of computing resources required for harnessing the potential of these NVMe devices. These consolidate the foundation for our motivation to design and develop *KV-SSD*, as an alternative to such key-value stores, thereby allowing applications to achieve higher throughput and low latency with minimal (or no) hardware changes and low host resource usage.

2.1 Resource Demands of NVMe SSDs

We measured the resource usage of a block-based NVMe SSD to understand the minimum amount of resources (compute) required to saturate a SSD. Using flexible I/O tester *fio*, we performed both synchronous and asynchronous I/O benchmarks[1], varying the request sizes, the number of threads and queue depths with and without a file system. In

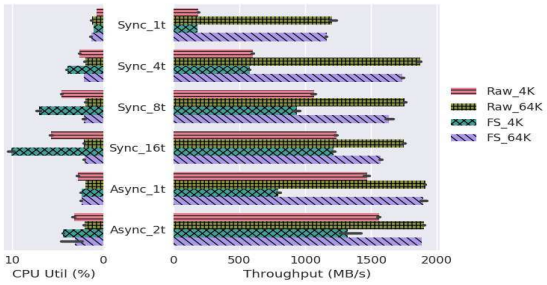


Figure 1: Sequential I/O benchmark (*fio* 50 GB write) on block NVMe SSD with and without ext4 file system.

Figure 1, we compare the throughput and CPU utilization for small (4KB) and large (64KB) block sizes; 100% utilization refers to the full utilization of all 48 cores of a NUMA node. We observe that saturating a device with synchronous I/O requires from 1.3 up to 5.8 CPUs while asynchronous I/O can fill the bandwidth with 1.0 to 2.0 CPUs. Usually, larger blocks are more efficient as they can fill the device bandwidth with fewer CPU resources and a smaller number of requests, although the average I/O latency increases [18]. Also, with a file system, the throughput is slightly lower in every case, even while using DIRECT I/O. Specifically, for the case of one asynchronous I/O thread (*Async_1t*), the throughput with file system experienced a huge drop compared to a raw device access, requiring one more CPU to saturate the device.

Takeaway 1: At least one CPU needs to be dedicated to saturate one NVMe device.

Takeaway 2: More CPUs are required as the processing overhead or the I/O hierarchy complexity increases. This is evident from the file system results discussed above.

Therefore, the additional CPUs needed to this ideal resource usage of 1 CPU per device can serve as a metric for scalability comparison of each key-value store.

[1]The experimental testbed setup is described in Section 4.

2.2 Conventional Key-Value Store

Many conventional Key-Value storage applications are specifically designed/optimized for NAND flash SSDs [1, 2, 9, 15, 16, 22]. The resource consumption and processing overhead varies with each key-value store based on the architecture and objective of the key-value store. Figure 2 illustrates the typical design of such key-value stores. These KV stores typically use one of three I/O paths to access the storage devices, as shown in Figure 2:

Path 1: Store user data on files and use the kernel file system, block cache, block I/O layers and kernel drivers;

Path 2: Bypass the kernel file system and cache layers, but use the block layers and kernel drivers;

Path 3: Use of user-space libraries and drivers.

We have chosen three representative popular host-side key-value stores following all three paths to illustrate the problems faced by the host-side key-value stores, which aid in the design and evaluation of our work.

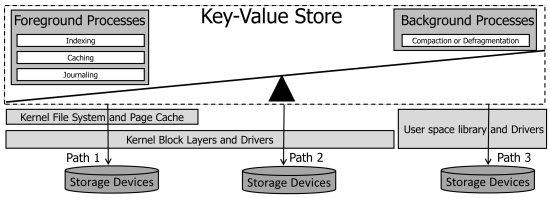


Figure 2: Processes in conventional Key-Value Stores.

RocksDB is a popular Log-Structured Merge (LSM) tree-based key-value store, optimized for fast, low latency SSDs and follows ***Path 1***. RocksDB buffers KV writes in a small fixed-size (default 64 MB) in-memory index table, called the *mem-table*, and writes another copy to a *write-ahead log* (WAL) for consistency. A configurable number of flush threads persists the *memtables* to stable storage. The default I/O request size of flush thread's is 1 MB. RocksDB has a background *compaction* process that re-organizes the user data files, which invokes many read and write operations, and competes with the foreground request processing threads for computation and I/O resources. Due to compaction, the performance of all workloads suffers, while the overheads associated to compaction is minimized for sequential key workloads.

Aerospike is a distributed NoSQL database and key-value store. Our work is based on a single node's data storage performance and the in-node scale up potential, we limit our scope to these aspects. Aerospike bypasses the kernel file system (***Path 2***) and utilizes native, multi-threaded, multi-core read and write I/O patterns. Writes are buffered in-memory and once full, the write-buffer is flushed to persistent storage. Writes to the SSDs are performed in large blocks and these blocks are organized on storage like a log-structured file system. The write buffer size is kept small (128 KB for SSDs) as opposed to RocksDB, since there is no separate log writes, and by default multiple write buffers (upto 64 MB) are queued in memory. As in any log-structured write scheme, over time the records on disk blocks gets invalidated by updates and deletes. A *defragmenter* process keeps track of active records and reclaims space by copying the active records to a different block. Aerospike performs best for an insert workload, irrespective of the key insertion order. A workload with updates/deletes will trigger the *defragmenter* and the associated read and write amplification affect performance.

RocksDB-SPDK is RocksDB with SPDK support, one that enables RocksDB to use the user space SPDK drivers and the associated SPDK Blobstore File system (BlobFS) to store its files (***Path 3***). BlobFS is based on SPDK blobstore, which owns and splits the entire device into 1 MB regions called *clusters*. Blobstore manages parallel reads and writes to *blobs*, which consists of *clusters*. Blobstore avoids the page cache entirely, behaving like DIRECT I/O. Blobstore also provides zero-copy, lockless, truly asynchronous reads and writes to the storage device. All other working aspects except the one's discussed above for RocksDB-SPDK is the same as RocksDB.

By default WAL is disabled in SPDK support of RocksDB, i.e., RocksDB-SPDK, and turning it on resulted in crashes during our tests. As WAL is integral to RocksDB's design, evaluating RocksDB-SPDK without the log would be incorrect. For extensive comparison, RocksDB-SPDK performance shows the impact of user-space drivers designed specifically for NVMe devices. We also compare single device performance RocksDB with and without WAL in our evaluation to make a educated guess about expected performance in production environments (refer to Section 5). Additionally, current RocksDB-SPDK environment is designed and hard-coded to use only one device with an entire database instance. We were unable to run with multiple devices or multiple instances on a single device, since the underlying SPDK device structure gets locked. Therefore, we modified the code to allow multiple devices and enable it to use multiple databases, one for each underlying device in an application with a shared environment supporting all underlying devices.

2.3 Challenges and bottlenecks

With our understanding of the resource consumption requirements and architecture of conventional KV-stores along with the internals of the I/O stack, we summarize the major challenges and bottlenecks below.

- *Multi-layer Interference:* Along the odyssey of data access, user read or write requests typically travels through multiple I/O layers before being submitted to the device [19], as shown in Figure 3 (a). With increasing complexity of the I/O hierarchy, the delays associated to request processing

increases and makes it more difficult to debug problems faced in production environments.

- *Resource contention:* As discussed earlier, key-value stores need to balance between several foreground and background processes. The foreground processes try to saturate the SSD bandwidth with incoming user requests while the background processes concurrently organize the written data without hindering the execution of foreground ones. In practice, however, for utilizing the device bandwidth, as the number of foreground processes increases, the workload on background processes also increases, which leads to slowing down or stalling of the system. This limits the node's scalability as more CPUs are dedicated to support smaller number of high performance storage devices.
- *Data consistency:* A write-ahead log WAL is generally used for data consistency. Though it provides consistency, WAL reduces user throughput by half as the total amount of data written doubles. Moreover, WAL generates sparse and low-bandwidth synchronous writes in the foreground. Even when buffered and written as large burst writes favored by SSDs, they are interleaved with key-value store specific computations, increasing the inter-arrival time of I/O requests and thus lowering the device utilization.
- *Read and Write Amplification:* In their pursuit of high performance, key-value stores introduce processes like garbage collection, compaction, and/or defragmentation. These significantly increases the read and write amplification as they need to read the written data back and process it. Large amount of buffer cache is also not quite helpful when there are many devices installed due to cache pollution. Further, this amplification is in addition to the internal read and write amplification caused by the SSD's garbage collection process. The amplification decreases the throughput and adversely affects the device lifetime.

In the next section, we describe our solution, KV-SSD, as an alternative which takes into account the challenges, requirements and architectural limitations imposed by the current I/O stack and provides a holistic approach to achieve both performance and scalability of high performance SSDs.

3 Design of KV-SSD

The architecture of conventional host-side key-value stores has evolved with the increasing performance of storage devices, but this development has been limited by the increasing demands for host system resources per device. Figure 3 shows the I/O path for three key-value stores namely, the conventional block SSD, KAML SSD [11], and our KV-SSD. We observe that with increasing layers of I/O hierarchy, the delay associated to I/O access also increases.

While KAML provides basic key-value interface, i.e. put and get with fixed-size 8 B key and variable-size values to applications but lacks practicality for modern data-center workloads. For example, variable-size application keys would still require additional logs and data structures for translation of keys in the host as shown in Figure 3 (b). Moreover, KAML also lacks grouping of key-value pairs via the device interface. Therefore, the benefits derived from key-value interface can be nullified due to key data management required in the host.

To mitigate the adverse impact of host-side key-value management, we develop and design Key-Value SSD (KV-SSD), which offloads the key-value management to SSD, taking into account such application requirements. A dedicated key-value management layer in each device can provide advantages of isolated execution as well efficiently harness the SSD internal system resources. Moreover, owing to a reduced semantic gap between storage devices and key-value applications, KV-SSD brings the following software architectural benefits, which are discussed below.

1. *Lightweight Request Translation*: Since variable-size key-value pairs are supported, key-value to block translation is no longer required. Applications can directly create and send requests to KV-SSDs without going through any legacy storage stacks, resulting in reduced request processing and memory overhead.
2. *Consistency without Journaling*: It is not required to maintain metadata of key-value pairs at host side, eliminating the need for journaling or transactions to store metadata and data together. Consistency of key-value pairs is now managed inside KV-SSD, using its battery-backed in-memory request buffers. Each key-value request is guaranteed to have all or nothing consistency in a device. Therefore, for independent key-value pairs, write-ahead logging (WAL) mechanism is not necessary on host-side.
3. *Read and Write Amplification Reduction*: The host-side read and write amplification factors of KV-SSD remain to be an optimal value of 1, because it does not require additional information to be stored for key-value pairs. Internally, because variable-size keys are hashed before being written to an index structure, they still can be considered as fixed-size keys. Therefore, internal read and write amplification factors in KV-SSD remained the same as that of random block reads and writes in block SSDs.
4. *Small Memory Footprint*: Regardless of the number of key-value pairs in a device, the host memory consumption of KV-SSD remains a constant. Its memory consumption can be calculated by multiplying the I/O queue depth by the size of a key-value pair and the size of a value buffer. For example, if the queue depth is 64 and the maximum value size is 1 MB, which is large enough for most modern NVMe SSDs, and internal data structure for each key-value pair is 64 B, the

total amount of memory required to support one device is around 4 KB for keys and 64 MB for values.

Additionally, KV-SSD allows key-value store applications to easily adopt a shared-nothing architecture as there is no main data structures to be shared between devices. By doing so, the scalability limitation from synchronization effects can be avoided. It can avoid lock contentions on shared data structures and also eliminate on-demand reads for looking up the metadata associated with a given request. To demonstrate these benefits, we design and implement a prototype device on top of an existing enterprise-grade Samsung NVMe SSD. On the host side, we provide an NVMe device driver that supports vendor-specific key-value commands and the API library that communicates with the driver.

3.1 In-Storage Key-Value Management

To realize the concept, we added support for variable-size keys and values to existing block-based SSD firmware. We designed KV-SSD to follow similar data processing flow as in the block firmware and meet its latency requirement to reuse its internal hardware resources and events designed for block requests, including interrupts, memory and DMA (direct memory access) engines. The code for each task is stored in SRAM, and the indices for the main index structure are stored in a battery-backed DRAM as with normal block-based SSDs. In this section, we discuss the working details of our prototype KV-SSD.

3.1.1 Command processing: Our key-value commands are currently implemented as NVMe vendor-specific commands. We include 5 native commands for practicability: *put*, *get*, *delete*, *exist*, and *iterate*, as shown in Figure 3. The *put* and *get* commands are similar to the write and read commands of block devices, with additional support for variable sized keys and values. The *delete* command removes a key-value pair. The *exist* command is designed to query the existence of multiple keys using a single I/O command. Finally, the *iterate* command enumerates keys that match a given search prefix, which is represented by a 4 B bitmask in KV-SSD.

The key-value requests are designed to be executed in a pipeline manner, passing requests between tasks running in each CPU as shown in Figure 4. To process a *put* request, for example, the command header is first fetched from the device I/O queue and passed to *Request Handlers*. Then, the communication between the device driver and the device is initiated to transfer the key-value pair from the host system. Once the data becomes ready in the device DRAM, *Request Handler* passes the request to an Index Manager. As shown in Figure 4, the Index Manager first hashes the variable sized key to fixed length key inside the device and stores in the local hash table, which is then merged to the global hash table for physical offset translation, and finally the Index Manager sends the key-value pair to flash channels. The Index Manager also takes the first 4 B of the key for bucketing into containers known as *iterator buckets*, which is used for grouping all keys with the same 4 B prefix.

Large variable-sized key Support: KV-SSD is designed to support both variable-sized values and keys. The support for variable-size keys provides the opportunity for applications to encode useful information such as name and type, and more importantly eliminates the requirement of maintaining additional logs and indices for name resolution, thereby simplifying the I/O stack, refer to Figure 3 (c). The key-value commands with large keys are implemented as two separate NVMe transfers in KV-SSD due to the limited size of NVMe command packets, which can slightly increase transfer latency.

Iterator support: Key-value applications often require group operations to maintain objects. This includes rebalancing, list, recovery, etc. To support these operations, KV-SSD provides support for iterators, which can *list*, *create*, or *remove* a group. Internally, all keys matching MSB 4 B key keys are containerized into *iterate bucket* as shown in Figure 4. This buckets are updated in a log-structured manner whenever *put* or *delete* operation is processed and periodically, cleaned up by GC, as described later.

3.1.2 FTL and Indexing: Block-based FTL is extended to support a variable-size key-value pairs. Traditional page-based or block-based mapping technique using LBA as a key cannot be used as a main index structure in KV-SSD as the range of keys is not fixed. We use a multi-level hash table for fast point query as a global index structure in KV-SSD. A global hash table is designed to have all key-value pairs in the device, and shared by all *Index Managers*. Each *Index Manager* has a local hash table to temporarily store updates to reduce a lock contention on the main table. This local hash table is associated with a bloom filter to reduce the number of memory accesses to the index structure on reads for quick membership checking.

3.1.3 Garbage Collection: The garbage collector (*GC*) in an SSD is changed to recognize the key-value pairs stored in a flash page and updated keys in the iterator buckets. During cleaning, GC scans the keys in flash pages, checks their validity by looking them up in the global index structure, and discards the key-value pairs that are already deleted. The victim selection and cleaning policy is the same as block firmware, with addition to managing the global hash table and iterator buckets.

3.2 KV Library and Driver

KV-SSD library provides a programming interface to user applications. It includes a wrapper for raw key-value command sets and extra features, such as memory management,

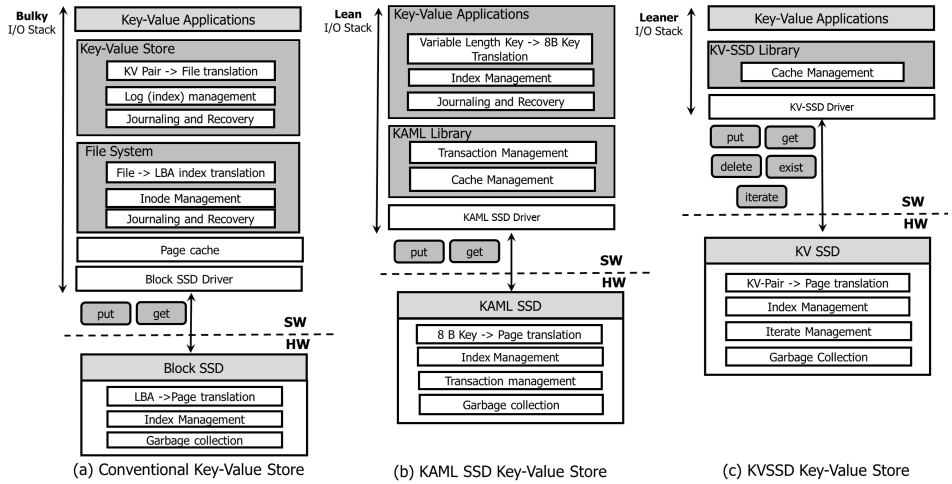


Figure 3: Comparative I/O storage stack for key-value stores based on (a) Conventional block SSD; (b) KAML SSD; (c) Samsung KV-SSD, respectively. More the layers in the I/O hierarchy, higher is the delay in data processing

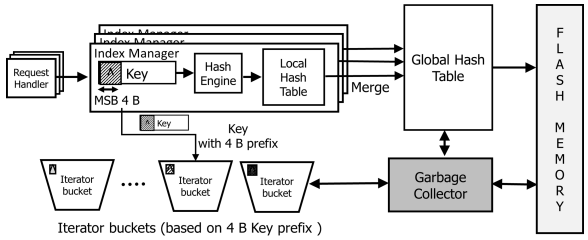


Figure 4: Data Flow in KV-SSD.

key-value LRU cache, synchronous and asynchronous I/Os, sorting, and a KV-SSD Emulator that mimics the behavior of KV-SSDs in memory. By default, KV library internally uses non-blocking, asynchronous I/O for performance. Synchronous I/O is simulated on top of asynchronous I/O using *mutex*.

Typically, the support for asynchronous I/O is provided in the block I/O layer in the kernel, but since KV-SSD applications bypasses kernel I/O stack, we moved the functionality to our KV driver. Whenever the I/O is completed, the KV driver stores the results of I/O in kernel memory and send a signal to event file descriptors. Then, users waiting for I/O completion using *select()* or *epoll()* will be notified.

KV-SSD also provides user-level NVMe drivers based on SPDK (Storage Performance Development Kit) [10], which supports memory management using *hugepage*s and asynchronous I/O APIs. Performance-wise, we have found no significant differences between the drivers. For example, while SPDK driver provides more control on assigning CPUs per its I/O queue, the multiple queue support in the latest kernels can provide the similar benefits. We chose to use the user-level driver in the evaluation for debugging purpose.

4 Experimental Setup

We use a custom designed storage node, *MissionPeak* with Ubuntu v16.04. The I/O bandwidth of one NUMA node is 24 GB/s, which can saturate 12-18 NVMe devices. The server is configured with abundant resources for scalability tests: 756 GB of memory, two Xeon E5 2.10Ghz, each having 48 CPUs with hyper-threading.

For multi-device configuration, we decided to use *sharding* rather than RAID. This is because we found that having multiple physical devices in one RAID device can suffer from I/O queue contention while *sharding* provides unique key-space for each key-value store, thereby ensuring isolation. Our experiment using 18 NVMe SSD with 144 instances of RocksDB also shows that sharding provides 3.4x performance, compared to one RAID-0 device that has 18 physical devices.

Existing benchmark programs for key-value stores, such as YCSB [7], are not designed for measuring the aggregated performance from multiple key-value stores running on multiple storage devices. This is crucial for our scalability evaluation as one instance of key-value store cannot saturate a NVMe SSD. Also, running multiple processes of the benchmark instead does not guarantee each process to be launched at the same time and executed concurrently.

We develop *KVSB* to coordinate the execution of multiple instances of key-value stores. *KVSB* assigns an available CPU to each key-value store and uses the memory from the same NUMA node. It has two phases of execution. In the initialization phase, all instances of a key-value store are created in separate threads and other resources are prepared such as memory pools. The execution phase starts when all instances becomes ready, running a combination of insert, update, read, or delete requests based on a given request distribution function.

5 Evaluation

We compare the performance and scalability of KV-SSD against conventional key-value stores using *KVSB*. Using the best performing and resource-efficient configuration for a single device as a baseline, the scalability of each key-value store is measured by increasing the number of devices and adjusting the configuration accordingly.

5.1 Resource Demands of Key-Value Stores

To understand the CPU requirement of key-value stores, we measure the normalized throughput by increasing the number of instances or internal flush threads until the device is saturated. The normalized throughput is calculated by dividing the number of key-value pairs written per second by the I/O bandwidth of the device. To minimize their processing overheads, 50 GB of sequential data is written, each consisting of 16B sorted keys and 4 KB values.

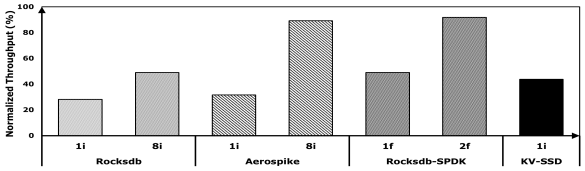


Figure 5: The amount of CPU resources required to saturate a NVME device. 'i' and 'f' are the number of instances and flush threads, respectively.

In Figure 5, we observe that conventional key-value stores require more than one thread to saturate a single NVMe device. RocksDB saturates the device with 8 instances, while the normalized throughput is around 50%. The rest of the device bandwidth is consumed to write on-disk WAL, which is as big as the entire workload (50%). By utilizing asynchronous I/O and disabling WAL, RocksDB-SPDK saturates the device with a single instance using 2 internal flush threads. Considering that a raw block device can be saturated with one thread in *fio*, the need for an additional flush thread can be seen as an overhead of its foreground processing. Aerospike also requires 8 clients to saturate the device because generated I/O concurrency per client at the server was not enough for an NVMe SSD.

However, KV-SSD saturates the device using only a single instance because it does not require any key-value processing in the host system. The benefits derived by bypassing legacy storage stacks and offloading the data management to the device is evident. Its normalized throughput, however, is currently limited to around 45% of the block device's maximum bandwidth in KV-SSD, due to the implementation limitations discussed in Section 7.

We used the best performing configurations for a single device and adjusted them accordingly to the total number of devices used and other resource constraints for our scalability experiments. Further, we examined if the performance scales linearly with increasing number of devices and the impact when more background processing is needed.

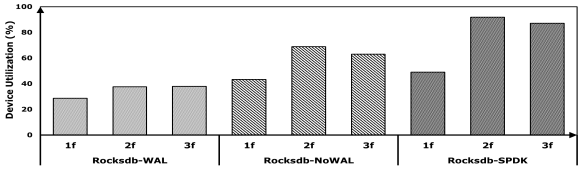


Figure 6: Sequential performance of Rocksdb and Rocksdb-SPDK with and without WAL.

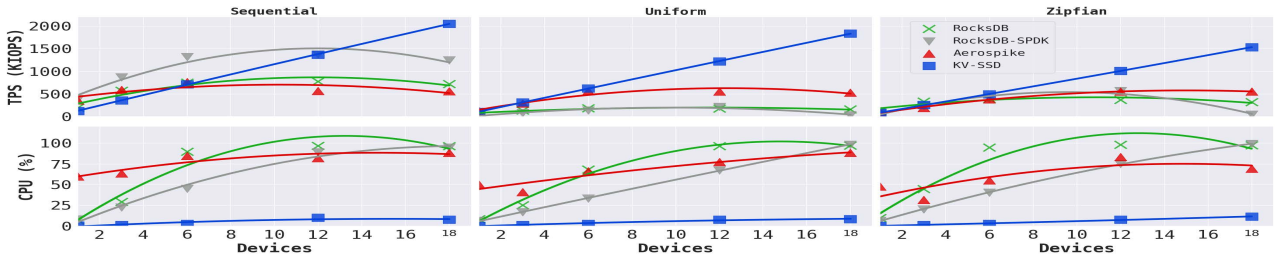
*5.1.1 SPDK Effects:*As mentioned in Section 2.2, although SPDK supports RocksDB, it has altered the design crucially by disabling WAL, thereby sacrificing recoverability. We measured the performance of RocksDB and RocksDB-SPDK with a different number of flush threads and different logging mechanisms, and the results are shown in Figure 6. Although all further experiments will have the RocksDB with WAL and RocksDB-SPDK without WAL, it is important to understand the impact of WAL on performance.

Compared to RocksDB-WAL, RocksDB-SPDK achieves higher disk bandwidth utilization, but we observe that most of the benefits is derived from disabling WAL because the difference between Rocksdb-NOWAL and Rocksdb-SPDK with 1 flush thread is marginal (5%). However, we noticed that the gap between RocksDB-NOWAL and RocksDB-SPDK becomes larger as we increase the no. of flush threads up to 3. The gap could come from the lightweight file system *BlobFS* that understands SSD performance characteristics, which uses asynchronous IO and large IO size unlike kernel file systems. We expect when WAL is enabled in RocksDB-SPDK, the overall performance of RocksDB-SPDK will drop by almost half (as workload size is doubled), and the performance gap between RocksDB and RocksDB-SPDK would become similar to between RocksDB-NOWAL and RocksDB-SPDK.

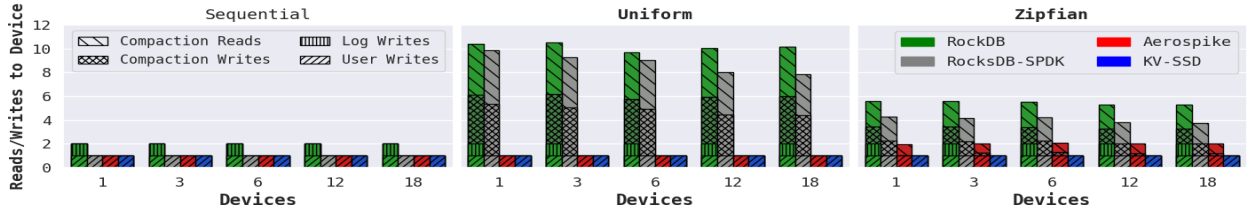
In order to use the best configurations that fully saturate the devices, for all further tests we use one instance of RocksDB-SPDK with 2 flush threads without WAL per device, and 8 instances of RocksDB with WAL per device.

5.2 Key-Value Store Scalability

To understand the potential of each key-value store to exploit device performance, we measure the throughput, CPU, and I/O utilization. We use 18 NVMe SSDs and assign multiple instances of the key-value store if one instance cannot saturate the device. We configured each conventional key-value store to get the maximum overall performance



(a) TPS and Host Resource Utilization



(b) Normalized IO

Figure 7: Write Performance of key-value stores. KV-SSD linearly scales while achieving an optimal write amplification factor of 1 and minimally using host-system resources.

separately for each experiment. The size of workload is approx. 192 GB of data, i.e.50M 16 B key and 4 KB value pairs.

5.2.1 *Workloads with Minimal Background Overheads:*

First, we measure the scalability of the foreground processing of each key-value store by using a sequential workload where unique keys are generated and sent sequentially, minimizing the need for post processing. The results are shown in the sequential workload column in Figure 7(a).

The performance of RocksDB and Aerospike is limited by CPU being a bottleneck when 6 or more devices are used. With 6 devices, there are 48 instances of RocksDB, each contains a active I/O thread, i.e. 48 I/O threads running on 48 CPUs with hyper-threading. After this point, its performance starts to degrade. Aerospike is configured to have 3 clients per device with one server, which showed the best scalable performance. It also saturates after 80% of CPU utilization.

RocksDB-SPDK shows better device and CPU utilization than RocksDB and Aerospike because of its asynchronous nature and disabled WAL, but it did not scale after 12 devices, due to CPU bottleneck. With 12 devices, as each RocksDB-SPDK instance uses 2 flush threads, it occupies all 24 physical CPUs with a total of 24 asynchronous flush threads.

On the other hand, KV-SSD linearly scales as the number of devices increases, requiring only around 6 CPUs for all 18 devices. This linear scalability is enabled by offloading key-value processing overhead to the device, conserving CPU and memory resources in the host system. While the current limited performance of KV-SSD as discussed in Section 3.1, only requires 0.5 CPU per device, we expect that 1 CPU per device would be necessary to saturate all 18 NVMe SSDs once the internal prototype issues between SSD hardware and KV firmware are resolved.

Figure 7(b) shows host-side write amplification of each key-value store while running the benchmark. The sequential workload column shows that the total amount of I/O is the same as the number of key-value pairs required by users for Aerospike, RocksDB-SPDK and KV-SSD. RocksDB incurs more I/Os because of its use of WAL. KV-SSD provides the same level of consistency as RocksDB without journaling because key-value pairs in this benchmark are independent. Another interesting observation from Figure 7(b) is that across all workloads the write amplification of KV-SSD is significantly lower and remains a constant.

5.2.2 *Workloads with Background Overheads:*

Unfortunately, an assumption of always sorted keys in the design of key-value stores is impractical, because keys can be hashed, overwritten, or deleted. Even if each user has an isolated key space and provides sorted keys, the performance of key-value stores can still be impacted because of reduced parallelism; each request needs to be processed in order. Therefore, utilizing the internal parallelism of storage devices becomes difficult. To explore the effects of such overlapping workloads, we measured the performance with keys that are generated using uniform and Zipfian distributions, which can contain 10-20%, 80% of duplicated keys, respectively.

To extract the best scalable performance for each workload, we benchmark each key-value store separately, varying their own performance parameters and selected the best performing ones. As a result, we configured RocksDB to have 2

background threads per instance (16 threads per device). For RocksDB-SPDK, we use 10 background threads per device up to 6 devices, and reduce it to 2 for a higher number of devices due to thread contention. Aerospike uses 3 clients per device, because its server cannot handle more requests than that. Finally, KV-SSD uses one thread per device.

In the case of uniform workload, as shown in Figure 7(a), we find that both RocksDB and RocksDB-SPDK suffer from the *compaction* overheads which require lots of background I/Os and memory, making foreground processes stall or slow down. The benefits of RocksDB-SPDK are nullified due to such processing overheads. On the other hand, Aerospike and KV-SSD retain their performance characteristics. This is because Aerospike does not differentiate sequential and uniformly distributed keys, and KV-SSD internally handles the overhead with its own resources. We also noticed that the write amplification of RocksDB-SPDK is lower than RocksDB, because by default it invokes compaction later than RocksDB.

With the Zipfian distribution, the amount of background overheads becomes lower than the uniform case. This is because many of overwrites can be cached in a write buffer, reducing the amount of writes, and the amount of compaction reduces because of duplicated keys. The defragmentation process of Aerospike also kicks in with this workload, increasing its write amplification.

Additionally, we measured the overhead of enabling a sorted key group in our KV library using the same workloads. Our evaluations shows that the cost of tree management including key comparison and re-locations is lesser than 1% of the I/O performance, and the memory consumption varies between 1 GB to 2 GB per device, depending on the distribution of keys. The write amplification factors increase to 1.18 and 1.09, for sequential and uniform workloads, respectively. This increase is attributed to store nodes to KV-SSD for fast initialization. These overheads of sorted key management can be kept low, because the index structure does not need any writes for consistency.

Overall, we observe that none of the conventional key-value stores can achieve linear scalability within a storage node, mainly due to CPU bottlenecks. While SPDK provides better CPU usage characteristics, but as long as host-side background processing is required, linear scalability would be difficult to achieve.

5.2.3 YCSB Workloads:

We measured the read and mixed workload performance of each key-value store using YCSB. YCSB-A and YCSB-B contains 50% read/50% write and 95% read/5% write, respectively. YCSB-C consists of 100% read operations and YCSB-D has 5% insert and 95% read operations. Zipfian distribution is used for keys except for YCSB-D, which uses latest distribution where most recently inserted records are in the head of the distribution [6].

Figure 8 shows that KV-SSD **without cache** scales linearly while providing comparable performance to conventional key-value stores with large cache because unlike other systems, KV-SSD uses its internal resources. Its read processing does not require additional I/O for metadata lookup, due to the battery backed DRAM inside the device.

When there are lots of updates (YCSB-A), the overall performance of conventional key-value stores is influenced by their background processing overheads as seen in Section 5.2.2. Other workloads with small overwrites shows similar performance characteristic as YCSB-C, which has only read operations. Since the workload distribution is either zipfian or latest, most of the operations are observed by memory. Aerospike uses its own cache and its performance is still limited after 6 devices similar to its write performance.

RocksDB, on the other hand, uses the OS page cache and scales well until 12 devices, but consumes lot of memory. RocksDB-SPDK performed poorly irrespective of amount of cache configured or configuration changes we did. On closer examination of internal statistics collected we noticed the file open, close and read operations had much higher latencies with RocksDB-SPDK than RocksDB. We also encountered many configurations where the system just hung without making any progress. As RocksDB-SPDK and BlobFS does not use the OS page cache, but instead manages memory by itself, we believe this is due to the system being in early stages of development and not being production ready yet.

5.2.4 KV-SSD with Cache:

We add 10 GB of LRU cache to each instance of KV-SSD and run YCSB-C, the overall read performance per KV-SSD instance increases by 4 to 6 times, as shown in Figure 9, and significantly outperforms all other key-value stores at fraction of memory utilization. To see the caching effect while isolating other performance impacting factors such as other inserts or updates, we show the results from YCSB-C (read only workload). Similar improvements can be expected in other workloads. It outperforms RocksDB and other systems by atleast 4 times and also consumes less memory (maximum memory consumption is approx. 180 GB for 18 devices, half of RocksDB memory consumption).

6 Related Work

NoveLSM [12] explores the benefits of NVMs for LSM-based KV stores by designing byte-addressable memtable and persistent skip list over NVM to reduce cost of compaction, logging, and serialization and increase read parallelism. HiKV [23] proposes to enable hybrid memory based persistent KV-store by constructing hybrid-indexing, i.e.

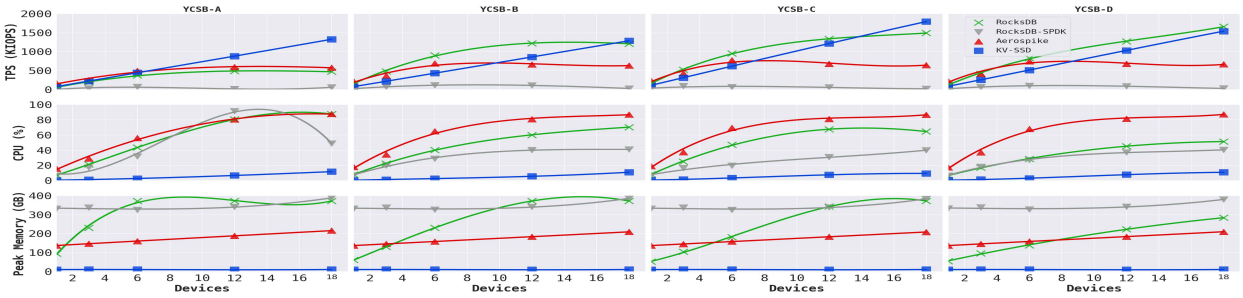


Figure 8: Read Performance of Key-value Stores. KV-SSD without a read cache shows compatible performance with conventional key-value stores that use a large read cache while also scaling linearly.

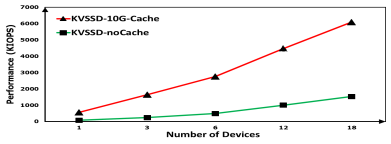


Figure 9: KV-SSD with a 10 GB read cache.

hash-index over NVM for fast searches and B+ Tree over DRAM for fast updates and range scans. NVMRocks [14] also proposes NVM-aware RocksDB by using byte-addressable persistent NVMs and tiers of storage hierarchy, i.e. read-cache, DRAM, NVMs, and SSDs to reduce loging overheads. While most of these object-stores optimize RocksDB to utilize NVMs for performance but the foreground and background processing still limits the CPU's scalability, while the objective of KV-SSD is to improve scalability and reduce the resource footprint with no infrastructural changes.

NVMKV [17] exhibits the benefits of FTL optimizations to map keys to address space in the device for KV store applications. BlueCache [24] presents a KV addressable flash design using FPGA to show the energy-efficiency and performance benefits of KV-SSDs. Seagate Kinetic Open Storage [3], provides key-value interface over Ethernet for data access residing in HDDs, rather than its native device interface, such as SATA and SAS. Our KV-SSD is the first prototype that runs on commercial SSD hardware and allows users to switch their block SSDs to key-value SSDs with just a firmware change. While KAML [11] also provides key-value interface but lacks practicality for modern data-center workloads.

While modern multi-core storage servers are capable of hosting multiple high-performance NVMe SSDs, software scalability issues prevents the utilization of all devices the hardware allows on a server. Flash storage disaggregation technique [13] shows the benefits of using detached flash tier for scalability, but it could shift the local resource utilization issue to the network congestion issue. User-space I/O stacks, such as *Storage Performance Development Kit (SPDK)*, are proposed to solve such scalability issues. The foundation of SPDK is an asynchronous, lockless NVMe user space driver that makes use of linux huge-pages for efficient TLB lookup.

Recently, the support for low-latency storage devices has been added to the Linux kernel 4.1 as well [5]. However, while it can improve I/O efficiency, it cannot solve the disk under-utilization issue due to the CPU contention issues between I/O and key-value store processes.

7 Discussion

The key-value functionality of KV-SSD was developed purely in firmware using the same hardware platform designed for block devices. This posed several engineering challenges including different design assumptions, limited compute resources, and hardware timing. As we optimize the firmware, we expect the performance of KV-SSD will eventually improve, being close to that of block devices. The key-value command protocols and KV APIs are currently under review by NVMe standard committee and SNIA [4], respectively. The source code of KV APIs, drivers, and *KVSB* are publicly available.

8 Conclusion

Through the performance analysis of three key-value stores in production using 18 NVMe SSDs, we show that high foreground and background processing of key-value stores prevent the performance and scalability of fast storage devices. Therefore, conventional key-value stores either exhibit suboptimal performance or limit scalability. To alleviate this trade-off, we developed KV-SSD and explored its use as an alternative to conventional host-side key-value stores. We show that by moving data management near the data, the hefty and redundant legacy storage stack can be simplified and the host resource consumption can be significantly reduced while saturating the device bandwidth without compromising data consistency. Through experimental evaluations, we show that KV-SSDs scale linearly with significantly lower memory, and CPU consumption, outperforming conventional host-side key-value stores.

References

[1] [n. d.]. Aerospike. https://www.aerospike.com/.

[2] [n. d.]. RocksDB. http://rocksdb.org/.

[3] [n. d.]. Seagate Kinetic Open Storage Platform: HDDs. https://www.seagate.com/tech-insights/kinetic-vision-how-seagate-new-developer-tools-meets-the-needs-of-cloud-storage-platforms-master-ti/.

[4] [n. d.]. The Storage Networking Industry Association : Advanced Storage Information and Technology. https://www.snia.org/.

[5] Alexander Gordeev. [n. d.]. blk-mq: Introduce combined hardware queues. https://lwn.net/Articles/700932/.

[6] Brian F. Cooper. 2010. YCSB. https://github.com/brianfrankcooper/YCSB/wiki/Core-Workloads.

[7] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC '10)*. ACM, New York, NY, USA, 143–154. https://doi.org/10.1145/1807128.1807152

[8] Tom Coughlin and Jim Handy. 2016. IOPS: Changing Needs. https://www.snia.org/sites/default/files/SDC/2016/presentations/performance/Coughlin-and-Handy_IOPS_Changing_Needs.pdf.

[9] Biplob Debnath, Sudipta Sengupta, and Jin Li. 2010. FlashStore: High Throughput Persistent Key-Value Store. *Proc. VLDB Endow.* 3 (Sept 2010), 1414–1425. Issue 2.

[10] Intel. [n. d.]. Storage Performance Development Kit. http://www.spdk.io/.

[11] Y. Jin, H. W. Tseng, Y. Papakonstantinou, and S. Swanson. 2017. KAML: A Flexible, High-Performance Key-Value SSD. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 373–384.

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

[13] Ana Klimovic, Christos Kozyrakis, Eno Thereska, Binu John, and Sanjeev Kumar. 2016. Flash Storage Disaggregation. In *Proceedings of the Eleventh European Conference on Computer Systems (EuroSys '16)*. Article 29, 15 pages.

[14] J Li, A Pavlo, and S Dong. 2017. NVMRocks: RocksDB on non-volatile memory systems.

[15] Hyeontaek Lim, Bin Fan, David G. Andersen, and Michael Kaminsky. 2011. SILT: A Memory-Efficient, High-Performance Key-Value Store. In *Proceedings of the 23nd ACM Symposium on Operating Systems Principles (SOSP '11)*. http://www.ssrc.ucsc.edu/PaperArchive/lim-sosp11.pdf

[16] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Hariharan Gopalakrishnan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2017. WiscKey: Separating Keys from Values in SSD-Conscious Storage. *Trans. Storage* 13, 1, Article 5 (March 2017), 28 pages.

[17] Leonardo Marmol, Swaminathan Sundararaman, Nisha Talagala, Raju Rangaswami, Sushma Devendrappa, Bharath Ramsundar, and Sriram Ganesan. 2014. NVMKV: A Scalable and Lightweight Flash Aware Key-Value Store. In *6th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 14)*. USENIX Association, Philadelphia, PA. https://www.usenix.org/conference/hotstorage14/workshop-program/presentation/marmol

[18] Pratik Mishra, Mayank Mishra, and Arun K Somani. 2016. Bulk i/o storage management for big data applications. In *Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS), 2016 IEEE 24th International Symposium on*. IEEE, 412–417.

[19] Pratik Mishra and Arun K Somani. 2017. Host managed contention avoidance storage solutions for Big Data. *Journal of Big Data* 4, 1 (2017), 1–42.

[20] Anastasios Papagiannis, Giorgos Saloustros, Manolis Marazakis, and Angelos Bilas. 2017. Iris: An Optimized I/O Stack for Low-latency Storage Devices. *SIGOPS Oper. Syst. Rev.* 50, 3 (Jan. 2017), 3–11.

[21] Woong Shin, Qichen Chen, Myoungwon Oh, Hyeonsang Eom, and Heon Y. Yeom. 2014. OS I/O Path Optimizations for Flash Solid-state Drives. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference (USENIX ATC'14)*. 483–488.

[22] Storage Performance Development Kit (SPDK). [n. d.]. RocksDB SPDK Integration. http://www.spdk.io/doc/blobfs.html.

[23] Fei Xia, Dejun Jiang, Jin Xiong, and Ninghui Sun. 2017. HiKV: A Hybrid Index Key-Value Store for DRAM-NVM Memory Systems. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. USENIX Association, Santa Clara, CA, 349–362. https://www.usenix.org/conference/atc17/technical-sessions/presentation/xia

[24] Shuotao Xu, Sungjin Lee, Sang-Woo Jun, Ming Liu, Jamey Hicks, and Arvind. 2016. Bluecache: A Scalable Distributed Flash-based Key-value Store. *Proc. VLDB Endow.* 10, 4 (Nov. 2016), 301–312.