FlashKV: Accelerating KV Performance with Open-Channel SSDs

JIACHENG ZHANG, YOUYOU LU, JIWU SHU, and XIONGJUN QIN,
Department of Computer Science and Technology, Tsinghua University

As the cost-per-bit of solid state disks is decreasing quickly, SSDs are supplanting HDDs in many cases, including the primary storage of key-value stores. However, simply deploying LSM-tree-based key-value stores on commercial SSDs is inefficient and induces heavy write amplification and severe garbage collection overhead under write-intensive conditions. The main cause of these critical issues comes from the triple redundant management functionalities lying in the LSM-tree, file system and flash translation layer, which block the awareness between key-value stores and flash devices. Furthermore, we observe that the performance of LSM-tree-based key-value stores is improved little by only eliminating these redundant layers, as the I/O stacks, including the cache and scheduler, are not optimized for LSM-tree's unique I/O patterns.

To address the issues above, we propose FlashKV, an LSM-tree based key-value store running on open-channel SSDs. FlashKV eliminates the redundant management and semantic isolation by directly managing the raw flash devices in the application layer. With the domain knowledge of LSM-tree and the open-channel information, FlashKV employs a parallel data layout to exploit the internal parallelism of the flash device, and optimizes the compaction, caching and I/O scheduling mechanisms specifically. Evaluations show that FlashKV effectively improves system performance by 1.5× to 4.5× and decreases up to 50% write traffic under heavy write conditions, compared to LevelDB.

CCS Concepts: • **Computer systems organization → Architectures**; **Firmware**; • **Hardware → External storage**;

Additional Key Words and Phrases: LSM-tree-based key-value store, open-channel SSD, hardware-software co-design, application-managed flash

ACM Reference format:
Jiacheng Zhang, Youyou Lu, Jiwu Shu, and Xiongjun Qin. 2017. FlashKV: Accelerating KV Performance with Open-Channel SSDs. *ACM Trans. Embed. Comput. Syst.* 16, 5s, Article 139 (September 2017), 19 pages.
https://doi.org/10.1145/3126545

1 INTRODUCTION

The log-structured merge tree (LSM-tree) [27] is one of the most popular data structures in key-value stores' implementations, since it converts the random updates into sequential writes. The LSM-tree caches the incoming updates in the memory buffer first and dumps the whole buffer to the persistent storage as one batch when the buffer is full. To accelerate read operations,

This article was presented in the International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS) 2017 and appears as part of the ESWEEK-TECS special issue.
Authors' addresses: J. Zhang, Y. Lu, J. Shu (Corresponding author), and X. Qin, Department of Computer Science and Technology, Tsinghua University, Beijing 100084, China; emails: zhang-jc13@mails.tsinghua.edu.cn, {luyouyou, shujw}@tsinghua.edu.cn, qinxj14@mails.tsinghua.edu.cn.

https://doi.org/10.1145/3126545

the LSM-tree keeps the data entries sorted in the hierarchy of multiple levels. When an LSM-tree level reaches its capacity, the data entries will be merged to the next level, called compaction. The dumping and merging operations of LSM-tree are all conducted in a log-structured way without overwrite. Due to the sequential update pattern, LSM-tree is designed to be HDD-friendly and is employed widely in large-scale production environments, including BigTable [6] and LevelDB [12] at Google, Cassandra [5] and RocksDB [11] at Facebook, HBase [15] at Twitter and PNUTS [8] at Yahoo!.

With the rapid development of NAND flash technology, the cost-per-bit of solid state disks is decreasing quickly. Due to the low access latency and high bandwidth, SSDs are replacing HDDs in many use cases. Commercial SSDs adopt flash translation layers (FTLs) to cover the limitations of NAND flash (e.g., erase-before-write and limited erase cycles), by introducing address mapping, garbage collection, wear-leveling, etc. To support the legacy storage systems, FTL abstracts the flash device as a block device to the kernel, which builds a semantic gap between the system software and the hardware device. LSM-tree is initially proposed to run on hard disk drives. Simply deploying LSM-tree-based key-value stores on SSDs is not efficient and may induce serious issues. The key-value stores and SSDs both have their unique features. However, the FTL and the file system block the awareness of each other and limit the communication through a set of POSIX APIs.

The unawareness between KV stores and SSDs not only fails to exploit various optimization opportunities, but also induces unnecessary overheads. We believe the following three issues are of major importance. First, there exists triple redundant and unnecessary functions in the LSM-tree-based KV store, file system and flash translation layer (FTL). Each of the layers maintains similar functions, including mapping table, garbage collection (GC) and space management. The no-overwrite update pattern of LSM-tree can ease the burden of the mapping table in the FTL. And the compaction process plays the similar role to the garbage collection in the file system. These redundant functions interfere with each other and induce heavy write amplification and performance decrease [25, 31, 34–36]. Second, the large degree of internal parallelism of SSDs has not been well-leveraged [1, 7, 16, 33]. Due to the abstraction of FTL, the unique properties of SSDs, including rich parallel access ability and unbalanced read/write performance, are not available to the key-value stores and the file systems. Third, the domain knowledge of LSM-tree has not been exploited, as the KV stores are considered as normal applications by I/O stacks in the kernel. Within the LSM-tree based KV stores, there exist distinguishing characteristics in the access pattern of different data files, and in the behavior of foreground and background threads. The I/O stacks and the FTL are unaware of these features and handle the requests from KV stores without specific optimizations. The root of these issues lies in the semantic gaps between KV stores and flash devices. Although the layered design in current storage architecture decouples the I/O stacks implementation, it induces the semantic isolation of applications and storage devices, which motivates us to consider the software/hardware co-design of flash-based key-value stores.

In this paper, we present FlashKV, a flash-aware LSM-tree-based key-value store running upon open-channel SSDs. The open-channel SSD is built on raw flash devices without a firmware flash translation layer [2, 20, 24, 28, 35, 36], and it exposes the hardware details to the applications, including the size of flash page and flash block, the number of flash channels. The basic idea of FlashKV is to use the LSM-tree-based KV store to manage the raw flash devices directly, bypassing the file system layer and FTL layer. The semantic isolation and the redundant management lying in the FTL and the file system are eliminated. With the device-specific information, FlashKV designs the parallel data layout which stripes the key-value files over multiple flash channels for parallel accesses. With the domain knowledge of LSM-tree structure, FlashKV adopts adaptive parallelism compaction, which limits the write parallelism of the compaction process so as to reduce the interference with foreground reads. FlashKV further optimizes the current I/O stacks,

including caching and scheduling mechanisms. The compaction-aware cache applies different eviction and prefetching policies to clients' data and compaction data for higher caching efficiency. And the priority-based scheduler is implemented to schedule the foreground requests prior to the backgrounds, in order to decrease the client-visible latency. Overall, FlashKV achieves up to 4.5× higher performance than LevelDB [12], and decreases 50% write traffic under write heavy conditions. Our major contributions are summarized as follows:

- We identify the unexpected inefficiency and overhead in current LSM-tree-based flash storage, and propose FlashKV, a flash-aware key-value store with a software and hardware co-design.
- We propose parallel data layout, adaptive parallelism compaction, compaction-aware cache and priority-based scheduler, which are specifically optimized based on the domain knowledge of LSM-tree and SSD.
- We implement FlashKV on a real open-channel SSD and compare FlashKV with LevelDB on different file systems, including open-channel based raw flash file system. Evaluations using various workloads show that FlashKV has higher performance with significant lower garbage collection overhead and less write amplification.

The rest of this paper is organized as follows. Section 2 describes the background and analyses the key issues in current flash-based LSM-trees which motivate our work. Section 3 describes the FlashKV design, including the system architecture, parallel data layout and optimizations in compaction, cache and schedule mechanisms. Evaluations of FlashKV are shown in Section 4. Related works are discussed in Section 5 and the conclusion is made in Section 6.

2 BACKGROUND & MOTIVATION

We first provide a short review of the implementation details on LevelDB, which our design is derived from. Then we discuss the critical issues lying in current flash-based KV store architectures, which motivate the FlashKV design.

2.1 LevelDB

LevelDB [12] is a popular LSM-tree implementation that originated from Google's BigTable [6]. LevelDB is the basis of many academic and industry works [11, 22, 32]. Figure 1 shows the architecture of LevelDB. LevelDB consists of in-memory data structures (Memtable, Immutable Memtable) and on-disk files (SSTable and metadata files). For the write queries (e.g., insert, update and delete), LevelDB first buffers them in the MemTable and writes the log in case of the crash. The key-value pairs in the MemTable are sorted by skiplist [30]. When the size of MemTable reaches a threshold, the whole MemTable is turned to be an Immutable MemTable, which will be dumped to the disk as one batch. A new MemTable will be created to buffer the next incoming writes.

After dumped from the memory, the key-value pairs are maintained in the files, called SSTable. The SSTables are organized in levels. Each level has different capacities. The higher the level is, the more SSTables it can contain. When the capacity of level L reaches its threshold, LevelDB starts the compaction procedure. It picks one SSTable in level L and the SSTables which contain overlapping keys in level $L+1$. Then LevelDB merge-sorts these SSTables and writes the newly generated SSTables to level $L+1$ in the background. The compaction procedure helps to keep the stored data in the optimal layout and makes sure the keys in the same level (except level 0) are stored in order, which helps to accelerate the lookups. In the lowest level, level 0, each SSTable is generated by dumping from the Immutable MemTable. The keys are sorted within each SSTables, but are overlapping among other SSTables in level 0. The compaction also ensures the fresher data are stored in lower levels. After compaction, the merged SSTables in level L and level $L+1$

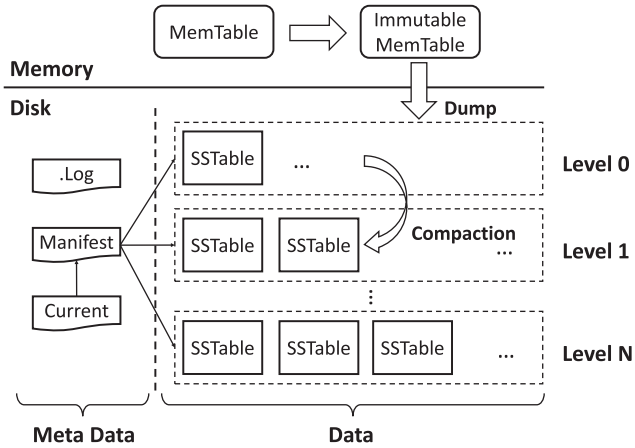
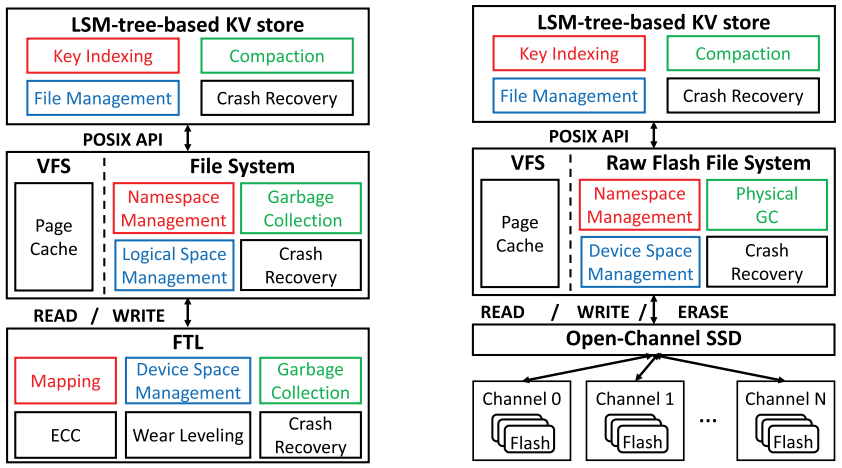


Fig. 1. Architecture of LevelDB.



(a) Key-Value Store over Commercial SSD (b) Key-Value Store over Open-Channel SSD

Fig. 2. KV Store Architectures on Flash Storage.

are discarded. The meta file "Manifest" records the valid SSTables after each compaction, and file "Current" records the current valid "Manifest".

For the read queries, LevelDB searches the MemTables, Immutable MemTables and SSTables from level 0 to level N in the sequential order, which ensures to get the latest version of the key-value pairs. LevelDB employs bloom filter [3] to determine whether the data exists in each level quickly.

2.2 Flash-Based KV Store Architectures

By introducing flash translation layers (FTL), Flash devices are seamlessly integrated into legacy storage systems. Most of the commercial SSD products are equipped with FTL. The easiest and common way to use SSDs to accelerate key-value systems is to run key-value stores upon SSD-based file systems, as shown in Figure 2(a). F2FS [19] is a hot choice, since it employs log-structured

update, and is designed to be flash friendly. However, this storage architecture remains critical concerns. For example, both the file system and the FTL adopt mapping, garbage collection, space management and over-provision mechanisms. These redundant functionalities (painted with the same color in Figure 2) not only bring inefficiency in I/O processing, but also induce significant write amplification under write-intensive workloads. Many works [20, 24, 31, 34–36] have been done to illustrate these issues. Another side effect of this architecture is the semantic gap brought by FTL. Recent file systems have revealed this problem, and propose different designs to bridge this gap [17, 18, 23, 24, 35, 37].

Open-channel architecture has been proposed [2, 20, 24, 28, 35, 36] recently to ease the critical issues discussed above. The Open-channel SSD exposes the detailed hardware information to the users. It employs a simplified driver, which only manages the hardware-related functions (e.g., ECC, bad block management), and leaves the flash space management, flash block recycling and internal parallelism control to the user-defined software. Based on the open-channel SSDs, raw flash file systems [20, 24, 35] have been designed and implemented to eliminate the redundant functions and semantic isolations between traditional file systems and the FTLs, as shown in Figure 2(b). However, there still remains several issues in deploying LSM-tree-based KV stores upon these raw flash file systems.

Redundant management overhead still exists between KV stores and file systems. Though raw flash file systems eliminate the redundant management between FTL and file system, the KV stores also maintain overlapped functionalities (i.e., space management, garbage collection) with file systems. The double mapping still exists, resulting from the key indexing in the KV stores and the namespace management in the file systems. The compaction procedure in LSM-tree acts similarly to the garbage collection threads in raw flash file systems. These redundant managements between KV stores and file systems induce inefficiency [31] and additional overheads [34] to the system.

The I/O stacks are designed for general purpose and are not optimal for KV stores. The domain knowledge of LSM-tree-based KV store has not been well leveraged by current file systems. The size of the key-value files (SSTable) in LevelDB is fixed, which is not leveraged in the space allocation and data layout by file systems. The cache of the file system is managed by page cache in the virtual file system (VFS) layer of the Linux kernel. The page cache is designed for general purpose, while the unique I/O patterns of LSM-tree are not considered in its eviction and prefetching mechanisms. In the scheduler of the I/O stacks, the requests are sorted only based on their basic properties, such as the direction of the requests (read/write), the size of the requests and their target block addresses. The scheduler is unaware of the priority of these requests in users' perspective, therefore it is unable to do further optimizations.

3 DESIGN AND IMPLEMENTATION

FlashKV is designed to eliminate the redundant management overhead by managing the open-channel flash device directly, bypassing the flash translation layer and the file system. FlashKV also optimizes the I/O stacks specifically, with the domain knowledge of both LSM-tree and open-channel flash device. To achieve these goals, FlashKV uses the following four key techniques:

- *Parallel Data Layout* to exploit the full parallelism of the bottom flash device when accessing the data, while maintaining low space management overhead.
- *Adaptive Parallelism Compaction* to accelerate compaction procedure under write-intensive workloads, and to reduce the writes interference with reads under read-intensive workloads.

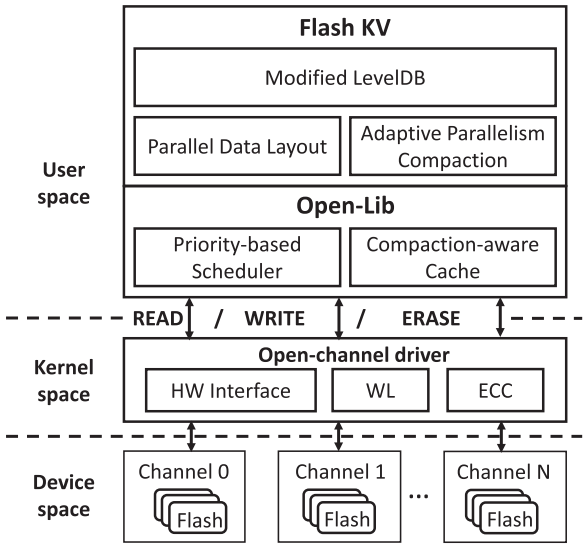


Fig. 3. The FlashKV Architecture.

- *Compaction-aware Cache* to optimize the caching algorithm by employing different management granularities and eviction policies for different I/O threads.
- *Priority-based Scheduler* to reduce the latency of the queries by assigning the clients' requests with higher priority.

In this section, we introduce the FlashKV architecture first, followed by the description of the above-mentioned four techniques.

3.1 The FlashKV Architecture

Due to the erase-before-write limitation of flash cells, FTL manages the flash space in a log-structured way [1, 29]. Since the LSM-tree also employs log structure to organize the key-value pairs in the persistent storage, it is possible to move LSM-tree-based KV stores upon raw flash devices with several modifications. FlashKV is designed based on LevelDB (one of the most popular LSM-tree based KV stores), and is deployed directly on open-channel SSDs, bypassing the kernel-space file system and device-space FTL, as shown in Figure 3.

FlashKV and the open-lib are running in the user space. FlashKV reuses most of the functionalities in LevelDB, but employs parallel data layout in space management and adaptive parallelism policy in compaction procedure. Like other user-space KV stores, the original version of LevelDB only manages the mapping between key-value pairs and the file names they stored. The KV pairs are located by two values, the pathname of the file and the offset within it. As for the file placement and the data layout among flash channels, they are taken care of by the file system and FTL. Since the FlashKV bypasses these two layers in its architecture, it manages the flash space in the user space directly. With the combination of the mappings in the key-value store, file system and FTL, FlashKV directly maps the key-value pairs to the raw flash pages in the open-channel device. FlashKV employs parallel data layout to fully exploit the internal parallelism of bottom SSDs. More details of flash space management and parallel data layout are given in Section 3.2. With the multiple flash channels exposed to the user space, FlashKV optimizes the performance of read requests under compaction, by employing adaptive parallelism policy. The policy limits the

number of flash channels used in compaction procedure under read-intensive workloads, so as to reduce the background writes interference with read requests. The details of adaptive parallelism policy are given in Section 3.3. FlashKV also combines the compaction procedure with garbage collection, due to the lack of FTL. After the compaction procedure generates new sorted files and deletes the old files, FlashKV sends erase requests to the bottom open-channel driver, to recycle the flash blocks which contain the deleted files.

The open-lib runs in the user space. It serves the I/O requests from FlashKV and dispatches these requests to the open-channel driver through the *ioctl* interface. The caching and I/O scheduling used to be implemented in the VFS layer and the generic block layer in Linux kernel for common storage systems. They are designed for general purpose, which is not optimal for LSM-tree-based KV stores. The target storage device of these implementations is HDD, which ignores the important features of SSDs, such as multiple parallel channels and unbalanced read/write latency. The open-lib redesigns and implements the cache and I/O scheduler in the user space. The caching and I/O scheduling algorithms are optimized using the domain knowledge of LSM-tree and SSDs, and they are described in details in Section 3.4 and Section 3.5.

The open-channel driver is designed and implemented in the kernel space. It maintains functions that are deeply related to the hardware, such as wear-leveling (WL) and ECC verification. It also translates the read, write and erase requests to the raw flash device through the hardware interface. The driver abstracts the raw flash device as a set of flash channels contained with multiple flash blocks. The structure of the raw flash device is described by 4 values, including the number of flash channels, the number of flash blocks within each channel, the number of flash pages within one flash block and the size of the flash page. The driver exposes these values to the user space FlashKV through the *ioctl* interface. With the information of the device structure, FlashKV assigns I/O requests with target channel ID and in-channel offset, which can specify their physical destination in the user space.

Compared to traditional storage systems, FlashKV moves most parts of the I/O stacks to the user space, and only keeps a thin layer, open-channel driver, in the kernel space. This design has several advantages. First, bypassing the file system and the FTL eliminates the overhead of triple management, and keeping the I/O stacks in the user space eases the overhead of frequent system calls and context switches. Second, the caching and I/O scheduling algorithm can be redesigned based on the unique features of LSM-tree and flash device. Third, the hardware details are exposed to the user space, which makes it possible to specifically optimize the data layout and compaction procedure in FlashKV.

3.2 Parallel Data Layout

Since the FTL and the file system are bypassed, FlashKV manages the raw flash space in the application level. There are two kinds of files in FlashKV, the SSTables that store key-value pairs and the metadata files (“Manifest”, “Current”, “.Log”) that used to record the validity of SSTables and to log the key-value pairs buffered in MemTable, as shown in Figure 1. The SStables are updated in a log-structured way, while the metadata files are updated in-place. FlashKV assumes the open-channel SSDs are equipped with a chip of non-volatile memory (NVM) or battery-backup DRAM [14, 32]. FlashKV stores the metadata files in the non-volatile memory zone of the device. These metadata files consume a small part of memory, as the log file is discarded after the MemTable is dumped to the device.

As for the SSTables, FlashKV employs the parallel data layout to store them on flash. FlashKV allocates and recycles the flash space in a super-block granularity. A super-block contains each one flash block from every flash channels, as shown in Figure 4. The size of SSTables in FlashKV is designed to be equal to the size of super-blocks, so as to store one SSTable in one super-block.

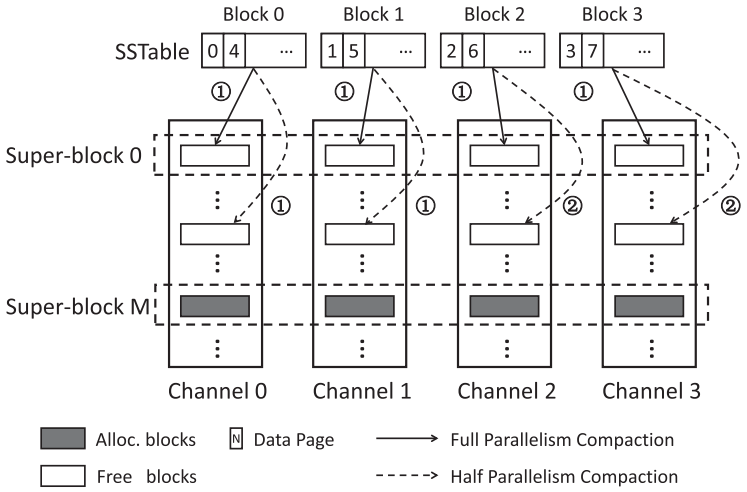


Fig. 4. The Parallel Data Layout of FlashKV.

Each super-block is assigned with a unique ID, which is represented by the offset of the blocks within the channel. Since all the SSTables are of the same size and they equal to the size of super-block, it saves a lot of efforts to manage them. The FlashKV only needs to maintain a mapping table between the SSTable's file name and the ID of the corresponding super-block. The index nodes per-file used in the file system are omitted. Due to the no-overwrite update pattern of the LSM-tree, the SSTables are written sequentially for only once (by dumping or by compaction). Therefore, the page-level or block-level mapping table used in the FTL is also omitted. Due to fewer mapping entries than page-level or block-level mapping, the super-block mapping table used in FlashKV stays relatively small even on large flash devices and can be entirely maintained in memory.

Besides the one super-block per-file mapping, the data layout inside each file is also parallelized. The data pages in the SSTable are striped over different flash blocks within the same super-block. Since the super-block consists of blocks from different flash channels, these flash blocks can be accessed in parallel. As shown in Figure 4, the neighbor pages are stored in parallel flash blocks. Therefore, sequentially accessing the SSTable on flash can be processed in parallel, which exploits the internal parallelism of flash devices. The parallel data layout doesn't induce additional overhead in page addressing. The target flash block and the offset inside the block can be simply calculated by modulo operations.

In implementation, the first flash pages and the last flash pages of the flash blocks in each super-block are reserved. Both of these two pages are used to store the metadata of the super-block, including the filename of SSTable stored in, the version of the SSTable and the state of the current super-block. The first pages are updated before the SSTable is written to the super-block. And the last pages are updated after the file writes finished. As the pages are written sequentially in each flash blocks, the first pages and the last pages are written with the SSTable file in one batch, which doesn't induce additional update overhead.

Crash Recovery. FlashKV also takes responsibility for the consistency maintenance of the whole system. After crash, the metadata files of the FlashKV can be recovered, as they are maintained in the non-volatile memory zone. As for the SSTables, FlashKV scans the last pages of each flash blocks and rebuilds the mapping table between SSTables and super-blocks. As the "Manifest" and "Current" are retained, the validity of the SSTables can be obtained. As the ".Log" is also

retained in the non-volatile zone, the latest updates before crash can also be restored. Furthermore, FlashKV also needs to detect all the half-updated super-blocks after crash. Due to the lack of the FTL, these written blocks need to be detected, which otherwise causes overwrite errors in flash memory. FlashKV scans the first page of each flash blocks and compares them with the last page of the flash block. If the data in these two pages doesn't match, then the flash block is half-updated. Since FlashKV updates the metadata ("Manifest" and "Current"), and deletes the merged SSTables only after the compaction finished, the data in half-updated flash blocks can be discarded directly. The whole super-blocks, which contain half-updated flash blocks, are erased and recycled after crash. In summary, FlashKV is able to maintain the consistency of the whole system after crash.

3.3 Adaptive Parallelism Compaction

As the write operation of NAND flash is 2× to 10× slower than the read operation [1, 13], the write requests could have a great impact on the read performance [1, 2]. When the flash chip is serving a write request, the following reads, which are dispatched to the same chip, will be blocked until the write finished. The latency of these read requests will be increased by 2× to 10×, due to the waiting for the write completion. On the contrary, if the write requests wait for the reads, the write latency increases slightly, due to the huge performance gap between flash read and write.

To reduce the write interference with read requests, FlashKV employs adaptive parallelism mechanism in compaction procedure. When the current workload writes heavily, the compaction procedure is in the critical path of system performance. In this situation, the compaction thread of FlashKV uses full parallelism to write the newly generated SSTables after merge-sort, so as to finish the compaction as soon as possible. Shown as solid lines in Figure 4, the key-value pairs in the newly generated SSTable are written to all the flash channels in parallel at the same time. At this point, all of the flash channels (four in the figure) are handling write requests from the compaction process, which exploits the full internal parallelism of the device. When the current workload is read-intensive, the compaction procedure is not in the critical path. The compaction thread limits the write parallelism for the newly generated SSTables, so as to decrease the interference with the read requests from clients. Shown as dotted lines in Figure 4, when the compaction thread decreases half of the write parallelism in the compaction, the key-value pairs, which belong to the first two flash channels, are written to the flash device first. After these write requests complete, the rest key-value pairs, which belong to other two flash channels, are written to the device in parallel. At any time of this writing process, only half of the flash channels are handling the write requests, and the reads on other channels will not be interfered by the compaction writes. Though the half parallelism compaction procedure consumes 2× time than full parallelism compaction, it induces little influence on the system performance due to the background compaction. Furthermore, the limited parallelism compaction helps to decrease the read latency, which in turn improves overall performance under read-intensive workloads. In all, the adaptive parallelism mechanism employs full write parallelism to accelerate the compaction under write-intensive workloads, and employs limited write parallelism to reduce the writes interference with reads under read-intensive workloads.

Besides the adaptive parallelism mechanism, FlashKV also adopts double threads for compaction. When the LevelDB is serving write-intensive workloads, plenty of SSTables are dumped to level 0 from the memory buffer. As LevelDB only uses one thread to perform compaction, these files can not be compacted to the next level quickly enough, which results in more and more files stacked in level 0. When the number of files in level 0 reaches the threshold (12 by default), LevelDB stops to serve the clients' write requests until the number drops. The single compaction thread used in LevelDB becomes the bottleneck of the performance under write-intensive workloads. FlashKV adopts two threads to speed up the compaction, which has been proved to be effective and efficient

in RocksDB [11] and LOCS [32]. Before compaction begins, the two threads select different SSTables, of which the keys are not overlapped. Then these two threads are able to perform compaction independently, without interfering with each other.

In implementation, FlashKV maintains a request window to determine whether the current workload is write heavy or not. FlashKV records the direction of clients' requests (read/write) between each compaction. If the percent of write requests is less than a threshold (30% in implementation), FlashKV decreases half of the write parallelism during compaction. Otherwise, the workload is considered to be write heavy and FlashKV uses full parallelism to write. There is one exception for level 0 and level 1. The SSTables in level 0 are dumped by Immutable MemTable, and they contain overlapped keys. To accelerate lookups in level 0 and avoid files stacking, FlashKV always uses full parallelism for compactions running at level 0 and the next level (level 1) in implementation.

3.4 Compaction-Aware Cache

In the original LevelDB stores, the I/O cache is maintained in VFS layer in the kernel space, called page cache. The cache is managed in memory page sized granularity with LRU algorithm. However, the page cache ignores the access characteristics of the compaction procedure in LSM-tree, which results in low efficiency. When the compaction procedure starts, it reads key-value pairs from selected SSTables for merge-sort. These key-value pairs are only used for once and are no longer needed afterward. However, they will be kept in the page cache for a long time, due to the LRU-based eviction. Furthermore, the prefetching of the cache can be more aggressive for compaction procedure, because all key-value pairs in the selected SSTables will be read for merge-sort. As the I/O stacks are bypassed in the FlashKV architecture, the open-lib redesigns the caching and evicting algorithms with the optimization for compaction procedure in FlashKV.

The open-lib separates the read requests from FlashKV into two types. The first type is clients' reads, which come from the "get" operations of clients. The Second type is compaction reads, which are issued by compaction threads for merge-sort. These two types of reads have different I/O patterns. For the clients' reads (get), the size of the requests is small, mostly within one memory page size. Due to the multi-level layout of the LSM-tree, the adjacent keys may be distributed among all the levels. Prefetching the neighbor page of clients' reads is not much helpful even under sequential accesses. For the compaction reads, the size of the requests is large, since the compaction procedure needs to read and merge-sort the whole selected SSTables. In FlashKV, the compaction threads read one batch (2MB in implementation) of the selected SSTables for merge-sort, and read the next batch after the merge-sort completes. Therefore, prefetching the next pages within the same SSTables is always effective for compaction reads.

Based on the I/O characteristics discussed above, the open-lib optimizes the caching algorithm with the compaction-aware mechanism, as shown in Figure 5. The open-lib reuses an LRU list to manage the cache space with some modifications. The caching granularity for the two types of reads is different. The clients' reads are managed in pages, while the compaction reads are managed in batches. The open-lib doesn't perform prefetching for clients' reads. As for compaction reads, the open-lib prefetches the next whole batch of data, after the first batch of SSTables are read by compaction threads. When the merge-sort on the read batch completes, the batch will be moved to the head of LRU list, as shown in Figure 5. Since this batch is no longer needed by compaction threads, it will be evicted as soon as possible. When the cache reaches its capacity, the open-lib evicts the head of the LRU list for recycling.

In implementation, FlashKV optimizes the I/O granularity of compaction procedure. The compaction threads read and write 2MB data in one batch, which largely decreases the number of requests comparing to 4KB in LevelDB. FlashKV adds tags in the read requests to pass hints to

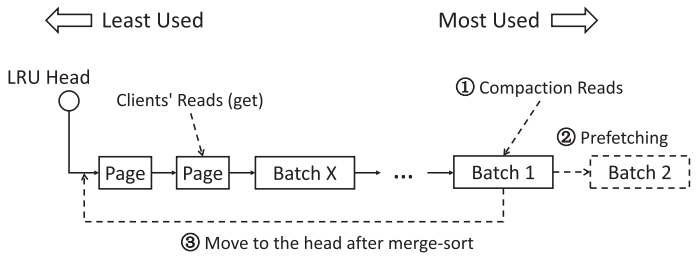


Fig. 5. Compaction-aware Cache.

open-lib. The tags contain the information about the source of the reads (from clients or from compaction threads) and the file name and offset of the requests. With these additional hints, the open-lib identifies the type of the read requests and applies compaction-aware caching. Besides the LRU list, open-lib also builds a radix tree [10] for each cached file to accelerate data indexing in cache. In the radix tree, the cached data page is indexed by its offset within the file. When the compaction threads are fetching the second batch of the data, the open-lib considers the first batch as useless and moves it to the head of LRU list.

3.5 Priority-Based Scheduler

In current I/O stacks, the I/O scheduler is implemented in generic block layer of Linux kernel. Due to the semantic isolation caused by block I/O interface, the I/O scheduler can only schedule the requests based on their basic properties, such as, the direction of the requests (read/write), the size of the requests and the logical block address of the requests. The scheduling algorithm is designed for general purpose, which fails to make specific optimizations for KV stores. For example, the requests from clients should be scheduled earlier than the compaction requests, which helps to decrease the user-visible latency. The erase requests which consume more time than other requests should be scheduled with different priorities based on the space utilization.

The open-lib designs and optimizes the scheduling in FlashKV based on priorities. The open-lib classifies the I/O requests into two groups, the foreground requests and background requests. The foreground requests are issued by clients for put and get key-value pairs. The background requests are issued by compaction threads for merge-sort. The scheduler is designed based on priority queue. It assigns foreground requests with higher priority than backgrounds. Within the same priority, the read requests are scheduled before write requests, due to the unbalanced read/write performance of flash devices. The scheduler in open-lib also needs to schedule the erase requests, which is different from the scheduler in the block layer. The priority of erase requests is determined by the number of free super-blocks in open-channel SSDs. When there's enough free space in the flash devices, the erase requests are assigned with low priority. Otherwise, the erase requests will be scheduled first for space recycling.

In implementation, the scheduler maintains a priority request queue for each flash channel, and the requests in each queue are scheduled independently. Before sent to the scheduler, the requests are added with additional flags to imply their owner. Based on these flags, the scheduler is able to classify the type of the requests and assigns the proper priority to them. For one exception, the compaction requests for level 0 and level 1, are classified to be foreground. They are assigned with the higher priority than other compaction requests, to accelerate the merge-sort in level 0, in case of files stacking. To avoid starvation, the priority-based scheduler employs deadline mechanism. Each request is assigned with a latest scheduling time. When the time expired, the request will be

Table 1. Parameters of the Testbed

Host Interface	PCIe 2.0 x8
Number of Flash Channel	34
Page Size	8KB
Block Size	2MB
Pages per Block	256
Read Bandwidth per Channel	94.62 MB/s
Write Bandwidth per Channel	13.4 MB/s

scheduled immediately. For erase requests, they are assigned the highest priority when the free space of the device drops below 20 percent. Otherwise, they are assigned the lowest priority.

4 EVALUATION

In this section, we evaluate FlashKV to answer the following three questions:

(1) How does FlashKV perform compared with LevelDB beyond different file systems, including open-channel based file system?
(2) How does FlashKV perform under heavy write traffic? And, how much write reduction can this architecture bring?
(3) What are the benefits respectively from the proposed techniques in FlashKV?

4.1 Experimental Setup

In the evaluation, we compare FlashKV with LevelDB on Ext4 [4], F2FS [19] and ParaFS [35]. Ext4 is a widely used Linux file system, from industry servers to mobile devices. Ext4 adopts in-place update and uses journal mechanism to maintain consistency. F2FS is a log-structured file system and it is specifically optimized for SSDs. ParaFS is a recently proposed file system running directly on the open-channel flash device without FTL. ParaFS employs a log-structured way to manage the raw flash space and the garbage collection, in the file system layer. In the original version of LevelDB, the size of SSTable is set to be 2MB by default. However, the size of SSTable in FlashKV is increased to match the size of super-blocks, which is larger than 2MB. For fairness concern, the size of SSTable in LevelDB is also increased. The other parameters of configuration in FlashKV and LevelDB stay the same. Through all the evaluations, the log of LevelDB and FlashKV is written in asynchronous mode (default mode in LevelDB), which means updating the log is not in the critical path of the system performance.

We run the evaluations on a customized open-channel flash device, whose parameters are listed in Table 1. To support the traditional file systems, like Ext4 and F2FS, we employ PBlk [2] on the device. PBlk is an FTL that designed for open-channel flash devices, and it is part of LightNVM [2] project which has been merged into Linux kernel since version 4.4. PBlk stripes writes over different flash channels in a page size granularity, to fully exploit the internal parallelism of the device. It allocates and recycles flash blocks in a super-block granularity, which is similar to FlashKV.

The experiments are conducted on an X86 server with Intel Xeon E5-2620 processor, clocked at 2.10GHz, and 8G memory of 1333MHz. The version of LevelDB used in the experiments is 1.14, and FlashKV is implemented based on the same version.

Workloads. The evaluations are conducted under five workloads of YCSB [9], summarized in Table 2. YCSB [9] is a framework designed to stress data serving systems. Since the structure of LSM-tree doesn't support range queries naturally, the LevelDB responses the range queries as plenty of lookups, which is very inefficient. The Workload E of YCSB, which is used to test

Table 2. Workload Characteristics

Name	Description	R/W	Dist.
Workload A	Update heavy workload	50/50	zipfian
Workload B	Read mostly workload	95/5	zipfian
Workload C	Read only workload	100/0	zipfian
Workload D	Read latest workload	95/5	latest
Workload F	Read-modify-write workload	50/50	zipfian

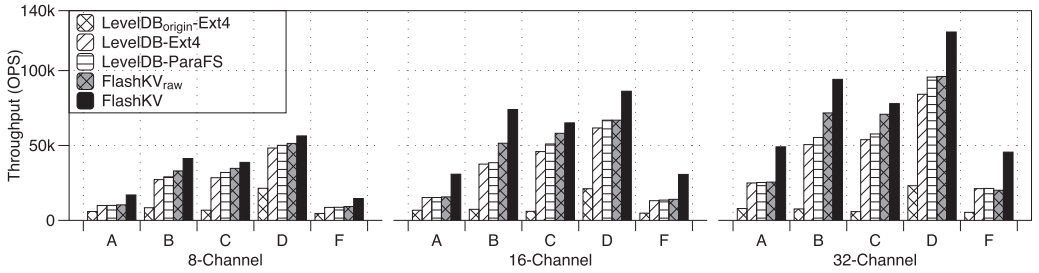


Fig. 6. Performance Evaluation (Light Write Traffic).

range queries, is skipped here. The YCSB contains two phases during each test, the load phase and running phase. The load phase inserts a large amount of key-value pairs to build a test database, and the running phase conducts put/get queries.

4.2 Evaluation with Light Write Traffic

In this section, we compare FlashKV with LevelDB on three file systems under light write traffic, without the interference of garbage collection. The capacity of the bottom device is set to be twice of the write size of the workloads to ensure no garbage collection in the FTL is involved. We want to show the pure performance improvement of the FlashKV over LevelDB, which benefits from the revolution of the architecture and the optimizations in I/O stacks. To show the system scalability with the internal parallelism of flash device, the experiments are conducted beyond various flash channels from 8 to 32. During each evaluation, YCSB inserts 10 million key-value pairs in the load phase, and then conducts 10 million operations (get/put) to the tested KV stores in the running phase. The compaction threads are invoked to maintain the optimal data layout during the test. We also conduct evaluations on the original version of LevelDB (LevelDB$_{origin}$) with 2MB SSTable, and on the raw version of FlashKV (FlashKV$_{raw}$), which adopts none optimizations in compaction, caching and scheduling.

Figure 6 shows the OPS (operations per second) of evaluated KV stores beyond different file systems. Through all the evaluations, the LevelDB performs similarly on Ext4 and F2FS. Due to the space limitation, we omit the results of LevelDB on F2FS. From the figure, we have two observations.

(1) **Performance.** FlashKV outperforms LevelDB and its original version in all cases, and achieves up to 2.2× and 13.3× improvement in the 32-channel case. The LevelDB$_{origin}$ performs poorly through all the test cases. The small size of SSTable causes the compaction involved frequently, which becomes the bottleneck of the performance. After extending the size of SSTable, the compaction performs more effectively and the performance of LevelDB is averagely improved by 3.7×. Since the PBlk stripes writes to different flash channels for full parallelism, LevelDB shows comparable performance on Ext4 and F2FS. ParaFS also uses full parallelism to send the write

requests. By eliminating the mapping and buffering in the FTL, ParaFS saves more memory space for caching, which explains the visible improvement in Workload D.

The FlashKV$_{raw}$ averagely achieves 20% improvement over ParaFS under random read workloads B and C, due to the shorter I/O path. Without optimizations for compaction procedure, the FlashKV$_{raw}$ performs similarly to ParaFS under write most workloads (A, F). FlashKV optimizes the compaction, caching and I/O scheduling with domain knowledge of LSM-tree and flash device. It averagely achieves 1.5× performance over raw version of FlashKV among three different channel cases.

(2) **Scalability.** With more channels, the flash device is able to handle more I/O requests in parallel, which provides higher bandwidth and lower read/write latency. Comparing three channel cases in Figure 6, FlashKV and LevelDB both have their performance improved when the number of channels is increased, but FlashKV improves faster. It outperforms LevelDB averagely from 1.5× in 8-channel cases to 1.8× in 32-channel cases. The LevelDB$_{origin}$ performs similarly beyond different flash channels, which certifies the bottleneck of the system performance lies in the software level.

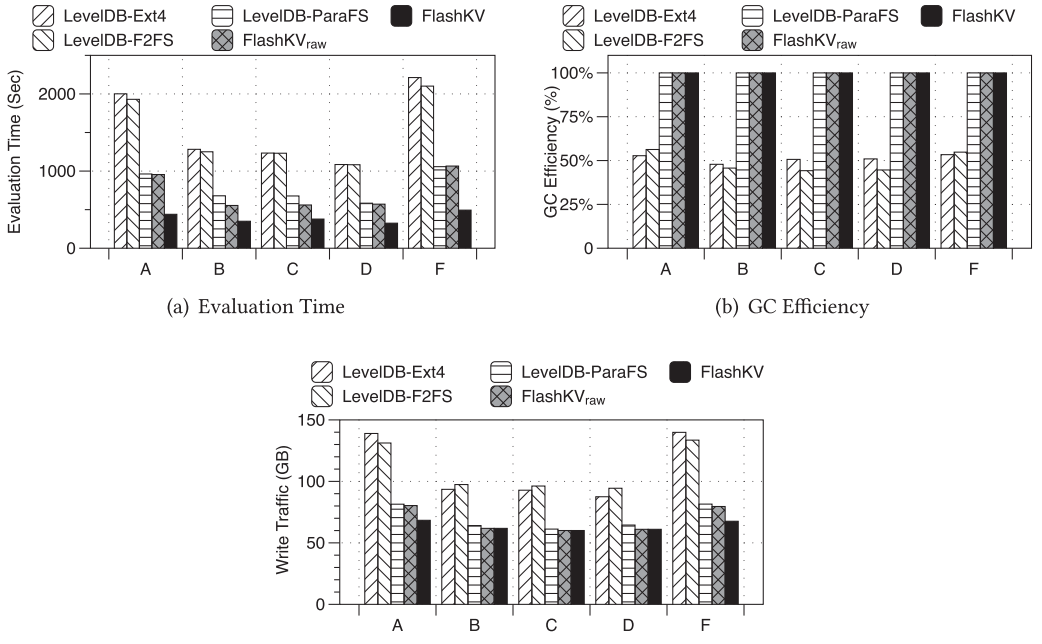
In all, FlashKV achieves the highest performance than two versions of LevelDB among all flash channel cases, when the write traffic is light.

4.3 Evaluation with Heavy Write Traffic

Since FlashKV manages the data placement and flash block recycling in the user space, we also evaluate its performance under heavy write traffic. In this evaluation, the sizes of write traffic in the five evaluated workloads are set to 2× ∼ 3× of the flash device capacity, to trigger the garbage collection frequently. During each run, we measure the time spent from loading the workload to running completion. The evaluations are conducted beyond 32 flash channels.

Performance. Figure 7(a) shows the time spent on each workload. We omit the original version of LevelDB, due to its poor performance. From the figure, FlashKV outperforms LevelDB on Ext4 and F2FS, by 3.2× ∼ 4.5×. The LevelDB on Ext4 and F2FS suffers from severe garbage collection overhead during the evaluation. Compared with open-channel file system, LevelDB on ParaFS, FlashKV achieves 1.8× ∼ 2.2× improvements. The FlashKV$_{raw}$ performs similarly to LevelDB on ParaFS, as they are nearly unaffected by the bottom garbage collection, which will be explained in details.

Write Traffic and Garbage Collection. To further understand the relationship between performance and heavy write traffic, we collect the statistics of GC efficiency and write size to the flash device during the evaluation. Figure 7(b) shows the GC efficiency of evaluated key-value stores during the experiments. The GC efficiency is measured using the average percentage of invalid pages in recycled flash blocks. FlashKV stores one SSTable file in one super-block, which contains 32 flash blocks in the 32-channel evaluations. Based on the LSM-tree structure, FlashKV only performs sequential writes to SSTables, and deletes the merged SSTables after the compaction finished. Therefore, the super-block can be directly erased after the deletion of the corresponding SSTable, without any page immigration. This explains the 100% GC efficiency in FlashKV and FlashKV$_{raw}$. As for LevelDB, the size of SSTable is also set to the size of 32 flash blocks and the compaction thread does the same work as in FlashKV. However, due to the abstraction of FTL, the SSTables in Ext4 and F2FS can not be stored aligning to the bottom flash blocks. As the FTL (PBlk) recycles flash blocks in the super-block granularity, the unalignment results in only 50% GC efficiency in average. By bypassing the FTL, ParaFS manages the data layout in raw flash device directly. The SSTable file in ParaFS is able to be aligned to bottom flash blocks. Since ParaFS recycles flash blocks in one block granularity, it achieves 100% GC efficiency when the SSTables are deleted.



(a) Evaluation Time

(b) GC Efficiency

(c) Write Traffic to Flash

Fig. 7. Evaluation under Heavy Write Traffic.

Figure 7(c) shows the write traffic to the device during the evaluations. LevelDB on Ext4 and F2FS writes 1.5× ∼ 2× more data to the flash device than FlashKV, due to the low GC efficiency. Averagely half of the flash pages in victim blocks need to be rewritten before erased. As the optimizations in caching, scheduling and parallelism control of compaction do not help to reduce the write size, FlashKV issues similar write traffic as FlashKV$_{raw}$ and LevelDB on ParaFS, under read-most workloads (B, C, D). FlashKV reduces 18% write traffic under write-intensive workloads (A, F). The reduction results from the double-thread compaction. As the SSTables in level 1 can be quickly merged to the next level, the compaction on level 0 will need to read less overlapped SSTables for merge-sort.

In all, FlashKV achieves optimal block recycling through data alignment. Compared to LevelDB on Ext4 and F2FS, FlashKV decreases the write traffic to the flash memory by 30% ∼ 50% and gains 3.2× ∼ 4.5× improvements. Compared to LevelDB on ParaFS, FlashKV decreases the write size by 18% and achieves 1.8× ∼ 2.2× higher performance under heavy write traffic.

4.4 Optimization Breakdown

With domain knowledge of LSM-trees and open-channel devices, FlashKV employs adaptive parallelism compaction, compaction-aware cache and priority-based scheduler to optimize its performance. In this evaluation, we show the performance improvement benefiting from each optimization individually. We use six versions of FlashKV. The FlashKV is the fully-functioned version with all three optimizations. FlashKV$_{raw}$ is the raw version of FlashKV that adopts none of the optimizations. FlashKV$_{LRU}$ replaces the compaction-aware cache algorithm with LRU. FlashKV$_{NOOP}$ replaces the priority-based scheduling with NOOP, which is the default scheduler for SSDs in Linux kernel. FlashKV$_{1thread}$ is the version with only one compaction thread, similar to LevelDB.

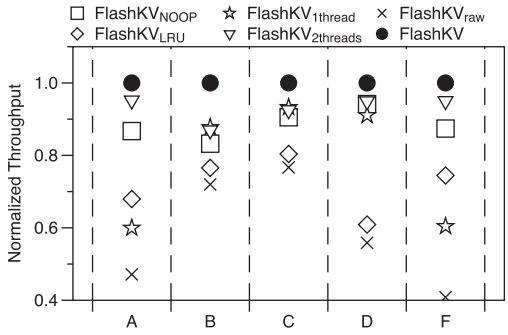


Fig. 8. Evaluation on Optimization Breakdown.

$FlashKV_{2threads}$ uses two threads for compaction without adaptive parallelism mechanism. The evaluations are conducted beyond 32 flash channels.

Figure 8 shows the performance of each run using six versions and the results are normalized to the fully-functioned FlashKV. To analyze the effectiveness of the compaction-aware cache, we compare $FlashKV_{LRU}$ with FlashKV. FlashKV outperforms $FlashKV_{LRU}$ in all workloads, from 1.25× to 1.64×. The compaction-aware cache optimizes the prefetching for compaction procedure and evicts the compaction batch as soon as possible which saves more space for caching foreground reads.

As for the priority-based scheduling, FlashKV performs up to 20% higher than $FlashKV_{NOOP}$. Giving higher priority to clients' requests helps to decrease the user-visible latency, as the compaction threads are working in the background during the evaluation. Since Workload D reads latest key-value pairs which are mainly stored in the memory, the performance couldn't benefit much from the optimization in scheduling. FlashKV only improves the throughput by 6% in Workload D, compared to $FlashKV_{NOOP}$.

Comparing the $FlashKV_{1thread}$ and $FlashKV_{2threads}$, the double-thread compaction outperforms by 58% in write-intensive workloads (A, F). As for the adaptive parallelism compaction, FlashKV performs 8% ∼ 15% higher than $FlashKV_{2threads}$ in Workload B and C. The limited write parallelism of compaction decreases the interference with the read requests from get operations, which improves the throughput under read intensive workloads. Since FlashKV uses full parallelism to compaction under write intensive workloads, it performs similarly to $FlashKV_{2threads}$ in Workload A and F. The improvement over $FlashKV_{2threads}$ in Workload D is less than B and C. Due to the latest reads distribution, most of the read requests are served in the memory.

In conclusion, the optimizations in FlashKV accelerate the compaction procedure, increase the memory utilization, decrease the writes interference with reads and improve the overall performance (1.8× over raw version of FlashKV).

5 RELATED WORK

Flash-based Architectures. OFSS [24] has compared different architectures of flash-based storage systems, and proposes the direct management of flash memory via software, which is later called open-channel SSD architecture. The open-channel architecture [2, 20, 28, 35] removes the FTL and exposes the bottom device information and control interface to users. SDF [28] abstracts each flash channel as individual devices to the applications. It is designed for web-scale storage, which needs large aggregated I/O throughput. ParaFS [35] and ALFS [20] propose to replace the traditional FTL with a simplified version. The simplified FTL manages the flash in a block [35] or

a super-block [20] granularity and leaves the garbage collection and address mapping to the file system. LightNVM [2] is a framework for managing open-channel SSDs in the software level. It designs the specification of the interface between users and open-channel devices. LightNVM has been merged into Linux kernel since 4.4. FlashKV adopts the open-channel driver to manage the hardware-related functions (e.g., ECC) and exposes the device information and control interface directly to the user-space KV store.

Some researches [14, 21, 24, 33] propose to use object-based SSDs to reduce the write amplification. OFTL [24] compacts the small updates under the byte-unit access interface, so as to reduce the write amplification from file systems. RWA [14] employs multi-level garbage collection and B+ table tree to mitigate the write amplification induced by partial page update and cascading update. FlashKV also reduces the write amplification from file systems by bypassing the file system layer. The write size in FlashKV is aligned to the flash page which avoids the partial page update, and the parallel data layout eases the burden of in-file indexing which eliminates the cascading update. OSSD [21] and p-OFTL [33] propose to separate the hot and cold data based on the I/O pattern and object attributes, for higher GC efficiency. FlashKV aggregates the data pages, which belong to the same file, to the same super-block. These pages will become invalid at the same time after the file is deleted, and FlashKV recycles the space without any page immigration.

Flash-based KV stores. Many works [22, 26, 31, 32] have been done to optimize the key-value stores, based on the features of SSDs. LOCS [32] also optimizes the LevelDB running on open-channel devices, SDF [28]. LOCS [32] writes each SSTable to one flash channel and exploits the device internal parallelism by increasing the number of concurrent file accesses. FlashKV stripes each SSTable among all flash channels, and could exploit full internal parallelism by accessing one file. The reads to the same SSTable can not be parallelized in LOCS as they are dispatched to the same flash channel, while they can be parallelized in FlashKV due to the parallel data layout. FlashKV also optimizes the compaction, caching and I/O scheduling algorithms with domain knowledge of the LSM-tree and flash devices. WiscKey [22] separates the keys from values stored in SSTables, so as to reduce the write amplification of values in the compaction. FlashKV also reduces the write amplification brought by garbage collection in file systems and FTLs. The optimizations in WiscKey are orthogonal to FlashKV's design. DIDACache [31] is a key-value cache system built on open-channel SSDs. DIDACache is designed and implemented based on hash table, and the data can be discarded directly for space recycling under the write-through policy. FlashKV is designed based on LSM-tree and focus on the persistent storage of the key-value pairs. NVMKV [26] is a hash-based key-value system that utilizes the advanced FTL capabilities (e.g., *atomic multi-block write*, *atomic multi-block persistent trim*) to provide scalability and ACID assurance. However, these advanced capabilities are not widely supported by current SSD products. FlashKV is designed based on the common structure of open-channel SSDs, which involves little hardware-related properties in the system design.

6 CONCLUSION

FlashKV directly manages the open-channel flash device in the user space. By bypassing the FTL and file system, the redundant management overhead and semantic gap between applications and hardware devices are eliminated. By employing the open-channel flash device, detailed hardware information is exposed to the application. With the domain knowledge of the flash device architecture and KV store, FlashKV designs the parallel data layout that exploits the internal parallelism of flash device. FlashKV also designs and implements the compaction, cache and I/O schedule mechanism with specific optimizations. We implement FlashKV on a customized open-channel flash device. Evaluations show that FlashKV outperforms LevelDB by up to 4.5×, and decreases the write traffic to the flash memory by 30% ∼ 50% under heavy write workloads.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their feedbacks and suggestions. This work is supported by the National Natural Science Foundation of China (Grant No. 61232003, 61327902, 61433008), the Beijing Municipal Science and Technology Commission of China (Grant No. D151100000815003), the China Postdoctoral Science Foundation (Grant No. 2016T90094, 2015M580098), and Samsung. Youyou Lu is also supported by the Young Elite Scientists Sponsorship Program of China Association for Science and Technology (CAST).

REFERENCES

[1] Nitin Agrawal, Vijayan Prabhakaran, Ted Wobber, John D. Davis, Mark Manasse, and Rina Panigrahy. 2008. Design tradeoffs for SSD performance. In *Proceedings of 2008 USENIX Annual Technical Conference (USENIX ATC)*. USENIX, Berkeley, CA.

[2] Matias Bjørling, Javier Gonzalez, and Philippe Bonnet. 2017. LightNVM: The linux open-channel SSD subsystem. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*. USENIX Association, Santa Clara, CA, 359–374. https://www.usenix.org/conference/fast17/technical-sessions/presentation/bjorling.

[3] Burton H. Bloom. 1970. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM* 13, 7 (1970), 422–426.

[4] MingMing Cao, Suparna Bhattacharya, Andreas Dilger, Alex Tomas, and Laurent Vivier. 2008. The new ext4 filesystem: Current status and future plans. In *Proceedings of the Linux Symposium. Ottawa, ON, CA: Red Hat. Retrieved.* 01–15.

[5] Cassandra. 2016. Apache Cassandra Documentation v4.0. (2016). Retrieved May 20, 2017 from http://cassandra.apache.org/doc/.

[6] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. 2006. Bigtable: A distributed storage system for structured data. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX, Berkeley, CA, 205–218.

[7] Feng Chen, Rubao Lee, and Xiaodong Zhang. 2011. Essential roles of exploiting internal parallelism of flash memory based solid state drives in high-speed data processing. In *Proceedings of the 17th IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 266–277.

[8] Brian F. Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. 2008. PNUTS: Yahoo!'s hosted data serving platform. *Proceedings of the VLDB Endowment (PVLDB 2008)* 1, 2 (2008), 1277–1288.

[9] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing*. ACM, 143–154.

[10] Corbet. 2006. Trees I: Radix trees. (2006). Retrieved February 28, 2017 from https://lwn.net/Articles/175432/.

[11] Facebook. 2013. RocksDB. http://rocksdb.org/. (2013).

[12] Sanjay Ghemawat and Jeff Dean. 2012. LevelDB, A fast and lightweight key/value database library by Google. (2012). Retrieved June 20, 2015 from http://code.google.com/p/leveldb/.

[13] Laura M. Grupp, Adrian M. Caulfield, Joel Coburn, Steven Swanson, Eitan Yaakobi, Paul H. Siegel, and Jack K. Wolf. 2009. Characterizing flash memory: Anomalies, observations, and applications. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. ACM, 24–33.

[14] Jie Guo, Chuhan Min, Tao Cai, and Yiran Chen. 2016. A design to reduce write amplification in object-based NAND flash devices. In *International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*. IEEE, 1–10.

[15] HBase. 2016. Apache HBase Reference Guide. (2016). Retrieved April 21, 2017 from http://hbase.apache.org/book.html.

[16] Yang Hu, Hong Jiang, Dan Feng, Lei Tian, Hao Luo, and Shuping Zhang. 2011. Performance impact and interplay of SSD parallelism through advanced commands, allocation strategy and data granularity. In *Proceedings of the International Conference on Supercomputing (ICS)*. ACM, 96–107.

[17] William K. Josephson, Lars A. Bongo, David Flynn, and Kai Li. 2010. DFS: A file system for virtualized flash storage. In *Proceedings of the 8th USENIX Conference on File and Storage Technologies (FAST)*. USENIX, Berkeley, CA.

[18] Jeong-Uk Kang, Jeeseok Hyun, Hyunjoo Maeng, and Sangyeun Cho. 2014. The multi-streamed solid-state drive. In *Proceedings of the 6th USENIX conference on Hot Topics in Storage and File Systems*. USENIX Association, 13–13.

[19] Changman Lee, Dongho Sim, Jooyoung Hwang, and Sangyeun Cho. 2015. F2FS: A new file system for flash storage. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST)*. USENIX, Santa Clara, CA. https://www.usenix.org/conference/fast15/technical-sessions/presentation/lee.

[20] Sungjin Lee, Ming Liu, Sangwoo Jun, Shuotao Xu, Jihong Kim, and Arvind. 2016. Application-managed flash. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*. USENIX Association, Santa Clara, CA, 339–353. http://usenix.org/conference/fast16/technical-sessions/presentation/lee.

[21] Young-Sik Lee, Sang-Hoon Kim, Jin-Soo Kim, Jaesoo Lee, Chanik Park, and Seungryoul Maeng. 2013. OSSD: A case for object-based solid state drives. In *Proceedings of the 29th IEEE Symposium on Mass Storage Systems and Technologies (MSST)*. IEEE, 1–13.

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

[23] Youyou Lu, Jiwu Shu, and Wei Wang. 2014. ReconFS: A reconstructable file system on flash storage. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST)*. USENIX, Berkeley, CA, 75–88.

[24] Youyou Lu, Jiwu Shu, and Weimin Zheng. 2013. Extending the lifetime of flash-based storage through reducing write amplification from file systems. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies (FAST)*. USENIX, Berkeley, CA.

[25] Youyou Lu, Jiacheng Zhang, and Jiwu Shu. 2015. Rethinking the file system design on flash-based storage. In *Communications of the Korean Institute of Information Scientists and Engineers (KIISE)*.

[26] Leonardo Marmol, Swaminathan Sundararaman, Nisha Talagala, and Raju Rangaswami. 2015. NVMKV: A scalable, lightweight, FTL-aware key-value store. In *USENIX Annual Technical Conference (ATC)*. 207–219.

[27] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. 1996. The log-structured merge-tree (LSM-tree). *Acta Informatica* 33, 4 (1996), 351–385.

[28] Jian Ouyang, Shiding Lin, Song Jiang, Zhenyu Hou, Yong Wang, and Yuanzheng Wang. 2014. SDF: Software-defined flash for web-scale internet storage systems. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, New York, NY, USA, 471–484. http://dx.doi.org/10.1145/2541940.2541959

[29] Xiangyong Ouyang, David Nellans, Robert Wipfel, David Flynn, and Dhabaleswar K. Panda. 2011. Beyond block I/O: Rethinking traditional storage primitives. In *Proceedings of the 17th IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 301–311.

[30] William Pugh. 1990. Skip lists: A probabilistic alternative to balanced trees. *Commun. ACM* 33, 6 (1990), 668–676.

[31] Zhaoyan Shen, Feng Chen, Yichen Jia, and Zili Shao. 2017. DIDACache: A deep integration of device and application for flash based key-value caching. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*. USENIX Association, Santa Clara, CA, 391–405. https://www.usenix.org/conference/fast17/technical-sessions/presentation/shen.

[32] Peng Wang, Guangyu Sun, Song Jiang, Jian Ouyang, Shiding Lin, Chen Zhang, and Jason Cong. 2014. An efficient design and implementation of LSM-tree based key-value store on open-channel SSD. In *Proceedings of the Ninth European Conference on Computer Systems (EuroSys)*. ACM, New York, NY, USA, Article 16, 14 pages. http://dx.doi.org/10.1145/2592798.2592804

[33] Wei Wang, Youyou Lu, and Jiwu Shu. 2014. p-OFTL: An object-based semantic-aware parallel flash translation layer. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE)*. European Design and Automation Association, 157.

[34] Jingpei Yang, Ned Plasson, Greg Gillis, Nisha Talagala, and Swaminathan Sundararaman. 2014. Don't stack your Log on my Log. In *USENIX Workshop on Interactions of NVM/Flash with Operating Systems and Workloads (INFLOW)*.

[35] Jiacheng Zhang, Jiwu Shu, and Youyou Lu. 2016. ParaFS: A log-structured file system to exploit the internal parallelism of flash devices. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*. USENIX Association, 87–100.

[36] Jiacheng Zhang, Jiwu Shu, and Youyou Lu. 2016. RFFS: A log-structured file system on raw-flash devices(WiPs). In *14th USENIX Conference on File and Storage Technologies (FAST)*. USENIX Association, Santa Clara, CA.

[37] Yiying Zhang, Leo Prasath Arulraj, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2012. De-indirection for flash-based SSDs with nameless writes. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies (FAST)*. USENIX, Berkeley, CA.

Received April 2017; revised May 2017; accepted June 2017