2019 IEEE 27th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)

iLSM-SSD: An Intelligent LSM-tree based Key-Value SSD for Data Analytics

Chang-Gyu Lee[†], Hyeongu Kang[†], Donggyu Park[†], Sungyong Park[†], Youngjae Kim[†,*]
Jungki Noh[‡], Woosuk Chung[‡], Kyoung Park[‡]
[†]Sogang University, Seoul, Republic of Korea, [‡]SK hynix
{changgyu, hyeongu, dgpark, parksy, youkim}@sogang.ac.kr, {jungki.noh, woosuk.chung, kyoung.park}@sk.com

***Abstract*—Several key-value stores such as RocksDB and MongoDB are implemented on the file system using the Log-Structured Merge-Tree (LSM-tree). The LSM-tree involves high compaction overhead. To minimize this overhead, WiscKey, the state-of-the-art LSM-tree, separates key and value, appends the value to the Value Log file, and LSM-tree manages only the key and Value Log offset. This minimizes the compaction overhead by reducing the number of SSTables managed by the LSM-tree. However, WiscKey still has a high I/O stack overhead that must go through the OS file system and block-layer. Therefore, this paper proposes *iLSM-SSD* that implements WiscKey in SSD and supports near-data processing. *iLSM-SSD* has the following features: (i) *iLSM-SSD* implements a key-value separation based LSM-tree in a limited memory space inside the SSD. (ii) The Value Log offset update management overhead incurred during the Value Log cleaning has a significant performance impact on CPU and memory-constrained SSD environments. To minimize this overhead, *iLSM-SSD* implements *Scattered Logging*, which reuses invalidated Value Log pages on the Value Log. (iii) *iLSM-SSD* manages the data layout internally. This enables *iLSM-SSD* to eliminate the need for file system interactions to obtain the data layout for in-storage processing on traditional block-interface-based SSDs. We prototyped the *iLSM-SSD* on the Cosmos+ OpenSSD platform in a Linux environment. Extensive evaluations with synthetic benchmarks have shown that the PUT performance of *iLSM-SSD* is 1.6-4 times higher than that of WiscKey implemented in RocksDB.**

***Keywords*-Log-Structured Merge-Tree, Key-Value Store, Solid-State Drive**

I. Introduction

A key-value store is a database that manages key-value pairs. Due to its simple key-value interface to access the value using the key, the key-value store has been widely employed in many modern applications [1]–[9]. For example, it can be used as a building block in applications, such as object stores in distributed object storage systems [1], database storage engines [2], [3], [5], [6], and caching systems [7]–[9] for big data storage systems. It can also be directly employed as distributed or local key-value database systems [10]–[14]. In particular, local key-value stores such as RocksDB [5] and MongoDB [6] are implemented using a Log-Structured Merge-Tree (LSM-tree) [15] on top of the OS file system.

[*]Corresponding author: Youngjae Kim

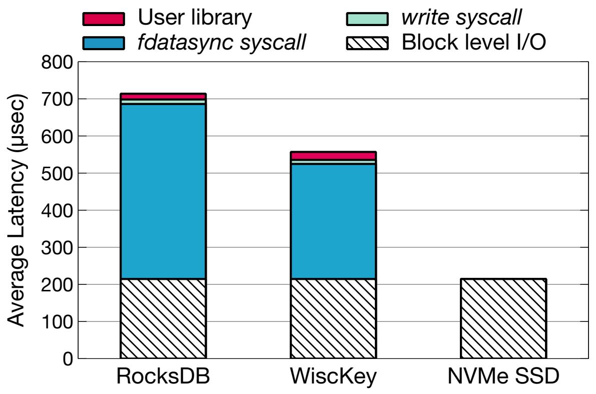


Figure 1. Comparison of latency breakdown of system calls for RocksDB, WiscKey, and NVMe SSD. The NVMe SSD was measured while issuing 4 KB writes directly on the NVMe SSD. Detail testbed configurations are presented in Table I, II.

The user can insert a key-value pair into the key-value store using a PUT request. In the key-value store using the LSM-tree, key-value pairs inserted are temporarily stored in the MemTable, an in-memory data structure. Once the MemTable reaches a certain threshold in size, it is flushed into persistent media in the form of Sorted String Table (SSTable), which is an immutable file. Since the SSTable is immutable, updating or deleting keys is handled as a PUT request with a new value or a special value indicating that the key has been deleted. Therefore, the old keys remain in the old SSTables. The user can retrieve the corresponding value of the key using a GET request. The key-value store using the LSM-tree first searches the key in the MemTable. If successful, it returns its value. Otherwise, it will search it in SSTables. Besides, the LSM-tree performs compaction. Compaction is the process of merging SSTables and sorting them to create new SSTables. The obsolete keys can only be reclaimed only in the compaction process because the SSTable is immutable. However, the compaction process incurs high write amplification because key-value pairs need to be re-written multiple times. This increases I/O traffic in the file system, which is a major cause of performance degradation.

WiscKey [16] has solved the aforementioned write amplification problem that occurs during the compaction of LSM-tree by using the key-value separation. In WiscKey, values are appended to the Value Log, and only the key and its Value Log offset are stored in the LSM-tree. This key-value separation significantly reduces the total size of SSTables and alleviates the write amplification problem that occurs during the compaction process.

However, WiscKey does not eliminate the I/O stack overhead because it is implemented in user-space and runs on top of the OS file system. In particular, when the strong consistency is to be assured, the key-value pairs must be persisted to the Write-Ahead-Log (WAL) or Value Log immediately on the SSD for every PUT request using system calls such as *fsync* and *fdatasync*. Therefore, *fsync* or *fdatasync* latencies of the file system directly affect the performance of the key-value store.

To confirm this overhead, we implemented the key-value separation technique from WiscKey in RocksDB (WiscKey). And, we compared it to the default RocksDB without the key-value separation (RocksDB). For comparative evaluation, we measured the average latency of RocksDB and WiscKey for 1 million unique 4 KB key-value PUT requests. To guarantee strong consistency, *fdatasync* is called for every 4 KB value PUT request, to persist the Write-Ahead-Log (WAL) in RocksDB and the Value Log in WiscKey. Figure 1 shows the time-break down of PUT requests by separating block-level I/O time from file system overhead. WiscKey shows a latency of 0.72 times lower than the default RocksDB by reducing write traffic due to the key-value separation. However, WiscKey still shows about 2.5 times higher latency compared to the conventional block-based NVMe SSD. Thus, in this paper, we attempt to build a Key-Value SSD which offers a key-value interface and implements the LSM-tree using the key-value separation within the SSD.

When analyzing the data stored in the key-value store, the user performs the following procedure – i) the value data of the key is loaded into the user-level memory via a GET request to the key-value store, and ii) the user uses the host machine's CPU and memory to run the data analysis kernel for the data. This involves data movement between the host and the SSD. To minimize this data movement cost between the host and the SSD, there have been several recent studies on near-data processing that run an analysis kernel near data in SSD [17]–[19]. These studies are based on block device interfaces such as NVMe SSDs for near-data processing. Before executing the data analysis kernel in the SSD, applications such as near-data processing analysis frameworks need to find the data layout of the file to be analyzed through the file system. This still entails file system stack overhead. On the other hand, key-value SSDs keep the data layout of the value inside the SSD, which eliminates the cost for finding the data layout from the file system.

This paper proposes *iLSM-SSD*, an SSD that implements the LSM-tree with the key-value separation and the in-storage processing framework in the SSD. *iLSM-SSD* thus enables near-data processing for data analysis in the SSD. The *iLSM-SSD* is designed and implemented with the following design challenges:

- **Storage protocol for key-value interface.** The existing SATA and NVMe protocols are intended for block-based storage devices. The key-value SSD requires a new protocol definition to communicate with devices that are key-value interfaces. For this, we extended the NVMe protocol and support key-value operations for *iLSM-SSD*.
- **NAND flash characteristic.** As the LSM-tree is implemented in the SSD, SSTables are to be written directly on the NAND flash. Since a NAND page requires to be erased for re-writing, the partial update consumes limited Program/Erase cycle of the NAND flash and is also critical in performance. As the Value Log offset is inserted to the SSTable in the LSM-tree using the key-value separation technique, the SSTable can be maintained in a small and fine-grained manner. Therefore, the SSTable can be easily aligned to the NAND page size. Thus, when implementing the LSM-tree, the key-value separation will benefit from it.
- **Constrained memory resource.** The internal memory capacity of an SSD is highly limited compared to the host. In this respect, we should ensure that MemTable, LSM-tree metadata and memory space necessary for Value Log cleaning are carefully fit in a limited amount of memory. We analyzed the memory space overhead of the MemTable and LSM-tree metadata for *iLSM-SSD* and designed an efficient free space management technique using the existing FTL mapping table for efficient Value Log cleaning.
- **Near-data processing capability.** Near-data processing enables to utilize computational resources in the SSD to reduce the data movement cost between the host and the SSD. However, it does not eliminate the file system interaction overhead. The user application still requires to import the file layout to get block information by interacting with the file system, then inform the SSD of the blocks on which the analysis kernel operates. In *iLSM-SSD*, the layout of the data is managed by the LSM-tree in the SSD. Thus, the file system interaction overhead can be eliminated. To support this, we design a near-data processing framework for *iLSM-SSD*.

We implemented *iLSM-SSD* on the Cosmos+ OpenSSD Platform [20], which is the development board for implementing SSD device. To show the effectiveness of *iLSM-SSD*, we compared *iLSM-SSD* against WiscKey. From extensive evaluations with synthetic benchmarks, we observed that PUT performance is 1.6-4 times higher than WiscKey, but GET performance is significantly lower. This low performance of the *iLSM-SSD* for the GET workload is due to the limited caching effect of the SSD. However, GET performance can be improved if user-level caching or kernel-level caching is implemented. We also witnessed the file system interaction overhead to find out the data layout of a file in block-based SSDs to support near-data processing is significantly high, whereas *iLSM-SSD* eliminates such overhead, especially when there is a search hit at the MemTable. Even in the worst case, its performance is comparable to a block-based SSD with the file system.

II. Background

A. Log-Structured Merge-Tree and Key-Value Separation

Log-Structured Merge-Tree (LSM-tree) is a data structure widely employed for the key-value store. LSM-tree delivers high throughput under write-intensive workloads by generating sequential writes from user's put requests through buffering and batching. LSM-tree consists of MemTable in volatile memory and immutable SSTable in persistent storage. MemTable temporarily stores key-value pairs in main memory, which are to be flushed. LSM-tree manages SSTables in multiple different levels and data flushed from MemTable are stored in SSTable at level 0. Every SSTable in the persistent storage covers a key range from the smallest key to the largest key in sorted order. Except level 0, all SSTables in each level have disjoint key ranges with each other. Each level has a threshold in size, and when it reaches the threshold, it triggers the compaction process. When compaction is triggered, it selects a victim SSTable from the level where the compaction process is triggered. Then compaction merges the victim SSTable with SSTables having an overlapping key range in the next level and insert new SSTables to next level. Since SSTables are immutable, keys that were overwritten or deleted are reclaimed during the compaction process.

Unlike the conventional LSM-tree, WiscKey [16] proposes the key-value separation, where value is appended to the separate Value Log. Then it stores the key with its Value Log offset in LSM-tree. As WiscKey separates values from the LSM-tree and stores actual values in the Value Log, it largely reduces the size of SSTables compared to conventional LSM-tree. As a result, WiscKey reduced a significant amount of data that has to be read and written in the compaction process. Since both key and value are written to the Value Log, WiscKey ensures strong consistency by persisting every Value Log entry before sending a response to the user.

Figure 2 depicts how the user's key-value is stored on a PUT request in the LSM-tree based key-value store with key-value separation. ① The key and value are appended to the Value Log, which is in the persistent media. ② A pair of the key and its Value Log offset is inserted to MemTable. If a power failure occurs at this moment, Value Log is utilized as write-ahead-log (WAL) to replay user's requests that were in volatile MemTable. After successfully persisting Value

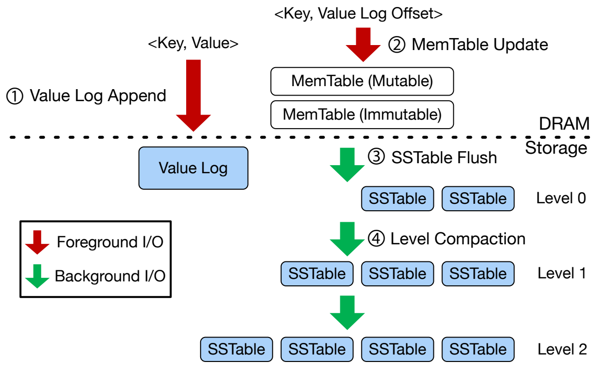


Figure 2. WiscKey's LSM-tree architecture and key-value insertion process.

Log and inserting to MemTable, a response to the user's request is sent. ③ When the size of MemTable reaches a threshold, LSM-tree marks the MemTable as immutable to prevent any further modification, then, converts it into an SSTable file and write it to Level 0. If the total size of SSTable in Level 0 reached the threshold, LSM-tree triggers the compaction process as a background job. ④ Compaction process will select a victim SSTable and merges it with SSTables in the next level. During this process, if any deleted or overlapping keys are found, they are removed from the SSTable. Compaction process could be recursively triggered if SSTables newly added to the next level make the total size of next level reaches to the threshold.

For a GET request, the Value Log offset corresponding to the key is required to get the actual value. LSM-tree searches the key in the following order: first at MemTable, second at immutable MemTable, and then at all the levels of SSTables. If the key is found in multiple different levels, the key at the lowest level has the latest Value Log offset. For example, if the key is found in level 0 and level 2, then the key in level 0 has the latest Value Log offset. After retrieving the Value Log offset successfully, the actual value is read from the Value Log to return to the user.

B. Value Log Cleaning

In a conventional LSM-tree where key-value separation is not employed, after the compaction process, deleted or overwritten keys and their values are removed from the LSM-tree. However, when the LSM-tree employs the key-value separation, their corresponding Value Log entries are not removed from Value Log until Value Log cleaning is triggered. During Value Log cleaning, invalid log entries should be removed and, at the same time, consecutive free space should be reclaimed for log entries to be newly appended. For Value Log cleaning, WiscKey appends both key and value to the Value Log and maintains the head and tail pointer of the Value Log. The Value Log cleaning is performed as follows. (1) From tail pointer, read a fixed-size

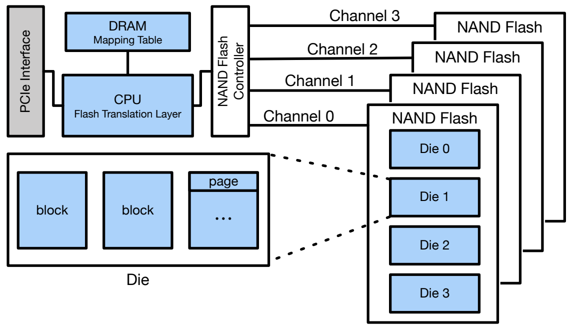


Figure 3. Architecture overview for an SSD.

chunk. (2) Search the keys inside the chunk from LSM-tree. (3) Compare current log offset and log offset generated from search results. If Value Log offsets are the same, the Value Log entry is valid. Otherwise, it is invalid. Valid log entries are appended at the head pointer whereas invalid log entries are deleted. After safely persisting valid log entries, a chunk from the tail pointer is truncated.

C. *Solid-State Drives (SSD)*

1) SSD Architecture: Figure 3 shows a simple example of the SSD internal structure. SSD consists of hardware components such as NAND flash chips, NAND flash controller, CPU and Memory. Unlike HDD, SSD runs software such as flash translation layer (FTL), garbage collection, and wear-leveling on the CPU using DRAM buffers. NAND flash employs page and block (a set of pages) as the units for I/O. Read and write are performed in the unit of NAND page and erase in the unit of block. In order to re-write a written page once before, erase must be implemented. NAND flash controller implements I/O operations for NAND flash chip. FTL running on the CPU provides an abstraction layer to the host by maintaining a mapping table that maps logical pages to physical pages. Since NAND flash does not allow overwrites, it uses the FTL to implement an out-of-place update operation.

2) I/O Flow: When a host requests a write to a specific logical page number (LPN), the FTL finds the corresponding physical page number (PPN) corresponding to the LPN in the mapping table. If the PPN mapping does not exist, the new free page is allocated to update the mapping information, and then the NAND flash controller performs I/O on the NAND flash channel corresponding to the PPN. If I/O is successfully performed, the SSD responds to the host. If there is a PPN corresponding to the LPN, an out-of-place update is performed by allocating a free page, and the previous PPN is made as an invalid page. Garbage collection is triggered if there is no free space. In this case, a victim block is selected, valid pages of the victim block are copied to the free block, and the LPN-to-PPN mapping information of the valid pages is updated accordingly. When the copying value pages of the victim block is completed, erase the block and mark all the pages that belong to that block as free pages. When a host requests a read to a specific logical page number (LPN), the FTL finds a corresponding physical page number (PPN) corresponding to the LPN in the mapping table. Then, it returns the data at that PPN to the host. If it does not exist, it means the LPN has never been written. Therefore, it will return some arbitrary data.

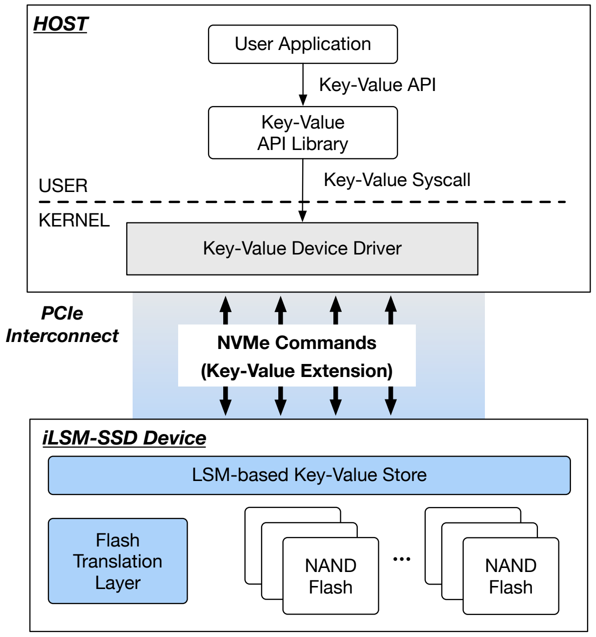


Figure 4. *iLSM-SSD* architecture and components.

In this paper, we implement the LSM-tree with the key-value separation by utilizing the CPU and DRAM of an SSD. The LSM-tree is a memory data structure that shares CPU and DRAM resources with FTL in the SSD. In particular, since the SSD's internal DRAM size is limited, the memory usage for the LSM-tree operation must be optimized for this small memory space.

III. Design and Implementation

In this section, we provide the design and implementation details of each component of *iLSM-SSD* in a top-down fashion – key-value API library, key-value device driver, NVMe protocol extension for key-value SSD, LSM-tree implementation in the SSD, and in-storage processing for data analytics.

A. *Overview for iLSM-SSD*

Figure 4 depicts an architecture overview of *iLSM-SSD*. *iLSM-SSD* consists of (i) a set of user-level key-value APIs, (ii) key-value device driver, and (iii) *iLSM-SSD* device. The key-value API library provides a set of key-value operations such as PUT, GET, and DELETE to the user. Key-value requests of the user are passed to the key-value device driver through system calls. The key-value device driver is a kernel module that is responsible for communicating with *iLSM-SSD* through the key-value device communication protocol. We designed the key-value protocol by extending an existing PCIe-based NVMe protocol. The key-value device driver delivers the user's key-value request to *iLSM-SSD*. Then, key-value commands are handled by the *iLSM-SSD* device.

B. Key-Value API and Device Driver

Key-value operations such as PUT, GET, and DELETE are implemented via system calls. These system calls pass these operational commands to the key-value kernel driver, which implements a key-value communication protocol to *iLSM-SSD*. Existing storage protocols such as SATA and NVMe were designed for block-based storage devices. Since the NVMe protocol is designed considering low latency and high parallelism of the modern SSD, the NVMe protocol becomes the most popular storage communication protocol for high-performance PCIe-interface SSDs. We carefully extended the NVMe protocol for the key-value protocol while taking full advantage of the existing NVMe protocol.

The basic operations of key-value interface are PUT, GET, and DELETE. We redefined key-value operational commands by using the areas defined in the NVMe commands such as vendor-specific OpCode, LBA start address, and reserved area. Figure 5 shows our newly defined NVMe commands for PUT, GET, and DELETE. In each command, the opcode area specifies these operations. The LBA start address area is used to specify an 8 bytes length key. Each NVMe command includes a page list to transfer physical addresses of pages on DRAM. Using these addresses, the SSD is able to pull data from the host for write command or push them to the host for read command. PUT and GET commands in the key-value extension of NVMe protocol utilize the page list for the value data of the key. However, for a key-value interface, the length of value data may not be expressed with the block size and the number of blocks because of variable value size. To solve the problem, we used some reserved area of the NVMe protocol to specify the length of value data for PUT command and the size of buffer for GET command. To implement key-value commands, we modified the NVMe device driver in Linux Kernel. All key-value commands are handled as same as any other NVMe commands and transferred through the PCIe interconnect.

C. LSM-tree based Key-Value SSD

To implement an LSM-tree based key-value SSD, we adopted the design idea of WiscKey [16]. It is because WiscKey offers small LSM-tree and less compaction overhead. The host-side implementation of WiscKey relies on the file system since SSTables and Value Log are stored as files. On the other hand, when WiscKey is implemented inside of an SSD, it has to deal with NAND pages rather than files because FTL does not provide the file interface. To reduce the semantic gap between LSM-tree and NAND flash, *iLSM-SSD* implements the WiscKey over the logical address space offered by the FTL. Thus, in *iLSM-SSD*, SSTables and Value Log are all managed in a log fashion on the logical address space of NAND pages.

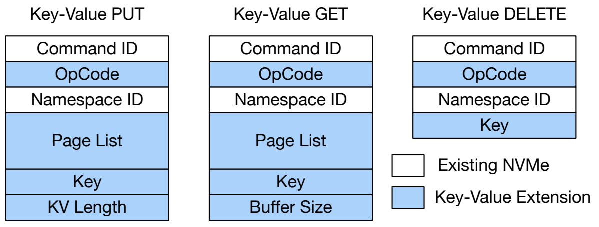


Figure 5. Key-value commands (PUT, GET, and DELETE) extending NVMe protocols.

Figure 6 shows the architecture of *iLSM-SSD*. It consists of LSM-tree, SSTable Log, Value Log and FTL. The FTL provides logical address space on top of physical NAND chips. The logical address space is partitioned into two areas – SSTable Log and Value Log. SSTables generated from the LSM-tree are stored in the SSTable Log area, and values are stored in the Value Log area. SSTable Log and Value Log provide free space through compaction and Value Log cleaning respectively.

An SSTable consists of a metadata block, bloom filter blocks and index blocks. Metadata of the SSTable includes SSTable id, SSDTable size, level it belongs to, location of the bloom filter and index blocks. Due to key-value separation, SSTable stores pairs of key and Value Log offset that are small and fixed. Thus, we embedded the Value Log offset corresponding to a key into index block. When searching a specific key in the SSTable, it first examines the bloom filter block. Then, it searches the key from index block for the Value Log offset. Note that the key may not be found in the index block due to false positive of the bloom filter. The metadata block of the SSTable is also cached in DRAM as per-SSTable metadata with maximum and minimum keys of the SSTable. Because SSTable searches involve reading multiple NAND pages, the maximum and minimum keys help filter out SSTables that do not need to be searched.

D. Memory Consideration

Since SSDs have a very limited internal memory capacity, it is crucial to evaluate the memory space overhead of *iLSM-SSD*. Compared to the in-memory data structure of the existing block-based SSD, the primary data structure added in the *iLSM-SSD* is MemTable for LSM-tree and per-SSTable metadata. Also, there is a reserved working space for holding SSTables during compaction and SSTable search.

1) Memory Overhead of MemTable: MemTable is one of the key data structures of LSM-tree. The input key-values are first stored and searched in the MemTable. Therefore,

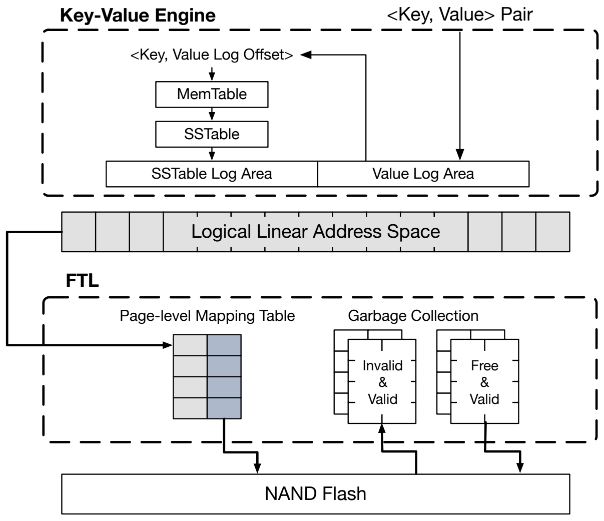


Figure 6. WiscKey implementation in an SSD.

size and number of MemTable directly affect to PUT performance. MemTable is usually implemented using a skip list. We can estimate the memory space overhead of a single MemTable from expected memory consumption of a skip list as the following example. A single node in the skip list has key, value and level pointers. The number of level pointers of each node starts from 1, and a level is added with probability p until it fails to add a level. In our design, we used $p = \frac{1}{2}$. Also we limited the maximum number of level as $MaxLevel$. We can get the expected number of pointers of a single list node in the skip list as $\sum_{i=1}^{MaxLevel} (\frac{1}{2})^i$. When $MaxLevel = 10$, the expected number of pointers of a list node is about 8. We use 4 bytes pointer, 8 bytes key, and 8 bytes Value Log offset in the skip list. When the skip list has 2,048 pairs of key and Value Log offset, the expected size of a full skip list is $(8B+8B+4B\times8)\times2048 = 64KB$. Since the memory space overhead of a single MemTable is small, the number of MemTables can be determined by the memory constraint of SSD.

2) Memory Overhead of per-SSTable Metadata: The memory overhead of per-SSTable metadata is dominated by the total number of SSTables in LSM-tree. With the key-value separation technique, the size of SSTable can be kept small compared to the conventional LSM-tree regardless of the value size. In other words, a single SSTable can cover more key-value pairs with the key-value separation. Thus, the same amount of key-value pairs can be managed with fewer SSTable. Also, it is essential to reduce the number of SSTables through compaction due to the characteristics of LSM-tree. Therefore, the memory space overhead of per-SSTable metadata is kept low through periodic compaction.

3) Memory Overhead of Working Area for Compaction and SSTable Search: Compaction and GET operation require working area in memory to load SSTables. Since invalid pages to be reclaimed by Garbage Collection of SSD are generated only after compaction, we can share memory space reserved for Garbage Collection of FTL with compaction of LSM-tree. For GET operation, only part of SSTable is loaded because bloom filter is examined first then index block is searched. So, we can limit the working space for the GET operation to multiple NAND pages.

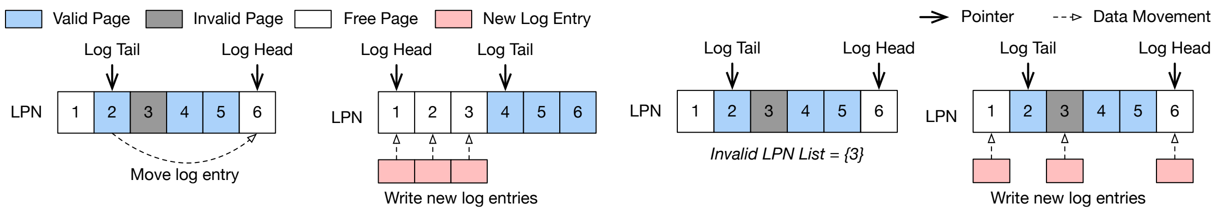
E. Scattered Logging

Value Log cleaning is required to reclaim invalid areas that occur in the middle of the log. In WiscKey, value and key are appended to the Value Log. And WiscKey maintains a log head pointer and a log tail pointer for the Value Log. The log head points to the location where the new entry will be written, and the log tail pointer points to the log entry where Log cleaning will begin. The detail of Log Cleaning is described in Section II-B. WiscKey stores key and Value Log offsets in LSM-tree, and locates Value Log entry using the offset. Therefore, if the Value Log entry is moved from the log tail to head during Value Log cleaning, the Value Log offset already stored in SSTables is needed to be updated. Due to the nature of LSM-tree, Value Log offset has to be inserted to LSM-tree again. However, this approach increases the compaction overhead because new offsets are added to the MemTable.

Compaction overhead caused by Value Log cleaning is fundamentally due to the offset change of Value Log entries. To minimize the offset change of Value Log entries during Value Log Cleaning, we propose the *Scattered Logging* by utilizing the mapping table of FTL already existing in SSD. A key idea of *Scattered Logging* is to reuse invalidated pages, the invalid LPNs, in the Value Log. Since pages in the Value Log are invalidated by update or deletion of keys, it is safe to record new Value Log entries. Therefore, the *Scattered Logging* allows writing new Value Log entries to the invalid LPN area and deferring the offset change of valid log entry during the Value Log cleaning.

To write new Value Log entry promptly in *Scattered Logging*, it is necessary to keep track of invalid areas (LPNs) scattered across the LPN space of the Value Log. Managing sparsely invalidated LPNs is challenging, mainly because of the size of the data structures required for tracking LPNs. Memory optimization is essential because the space cost of DRAM in SSDs is very high. For example, consider a list of invalid LPNs for the Value Log in 1 TB SSD with 16 KB page size. Assuming that each list node has an LPN and a pointer to the next list node, and each occupies 4 bytes, the maximum size of the list will be 512 MB. The maximum size is the case that all LPNs are in the list. However, in *Scattered Logging*, we utilize an existing FTL data structure to manage invalid LPNs without any additional spatial overhead.

The page-level mapping (FTL) already manages LPN-to-PPN mappings. The PPN field corresponding to invalid LPN in FTL mapping table can be used as pointers for invalid



(a) Value Log Cleaning (b) Scattered Logging

Figure 7. Design comparison on Value Log Cleaning and *Scattered Logging*.

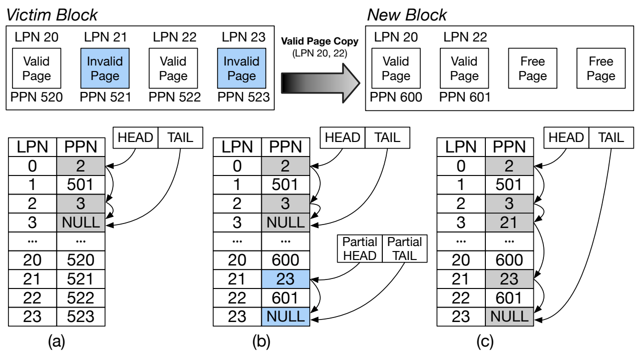


Figure 8. Free LPN List update with Garbage Collection.

LPN list management. This is because the invalidated LPN's PPN field is no longer valid. *Scattered Logging* manages the scattered invalid space by managing the invalidated LPNs as a linked list with corresponding PPN fields. Each PPN field has a next invalid LPN. Therefore, the invalid LPN list can be managed by only two additional variables in memory, head and tail pointers.

Figure 7 illustrates the difference between Value Log cleaning and *Scattered Logging*. Suppose we want to insert three log entries, but there is no free space and value log cleanup is called (Figure 7(a)). A valid Value Log entry at LPN_2 is moved to LPN_6 and Log Tail is updated to LPN_4. After that, new log entries can be appended to log head consecutively. Since LPN_2 is moved to LPN_6, corresponding Value Log offsets in LSM-tree are need to be updated. On the other hand, in the *Scattered Logging* (Figure 7(b)), invalid LPNs (which are actually free to be written) are managed by the invalid LPN list using PPN fields of the page-level mapping table. Thus, new log entries can be written at LPN_1, LPN_3, and LPN_6. Since no LPN's data needs to be moved, *Scattered Logging* does not require updates in LSM-tree for any other than new Value Log entries.

Figure 8 details how the PPN fields of the mapping table are used for *Scattered Logging* while the GC is running. In the Figure 8, there are two blocks, victim block and new block. During the GC, valid pages of the victim block are to be copied to new block. Figure 8(a) shows the invalid LPN list with head and tail pointers. In the example, during GC, two physical pages at PPN_{520}, PPN_{522} are copied to new block at PPN_{600} and PPN_{601}. In the mapping table, the PPN field of the corresponding LPN (LPN_{20} and LPN_{22}) is automatically updated. On the other hand, PPN field in the mapping table at LPN_{21} and LPN_{23} become reusable, as their physical pages corresponding to PPN are to be erased during the GC. Thus, these fields are managed by partial head and tail pointers (Figure 8(b)), and they will end up being merged with the head and tail pointers of the invalid LPN list (Figure 8(c)).

However, if the size of a single Value Log entry is larger than the page size and consecutive invalid LPNs are not found from the invalid LPN list, the valid Value Log entries have to be copied to make consecutive free space as conventional Value Log cleaning.

F. iLSM-SSD Operations

PUT, GET and DELETE operations of *iLSM-SSD* handled as follows:

1) PUT Operation: First, the user application issues PUT request using the Key-Value API library. The Key-Value API library uses the system call to pass PUT request to Key-Value SSD Device Driver and copy the key and value to the kernel space. The Key-Value SSD Device Driver packs the corresponding PUT request into the NVMe PUT command, inserts the kernel buffer page addresses into the page list, and sends it to the *iLSM-SSD* device via PCIe. *iLSM-SSD* fetches the PUT command, checks the page list, and copies the value into the internal memory using DMA. The value is appended to Value Log then the key, and the corresponding Value Log offset is pushed to MemTable. If the MemTable is full, MemTable is switched to Immutable MemTable and then flushed. In MemTable flushing, bloom filter and index block are created and packed into NAND page-aligned SSTable. After that, SSTable is inserted into SSTable log. At this point, NAND Flash Write is sent to NAND Flash Controller asynchronously. Then the per-SSTable metadata is updated and sends a response for the PUT command.

The flushed SSTable may trigger the compaction on level 0. Following that, the rest of level also may trigger another compaction. Our *iLSM-SSD* performs all compaction until there is no level triggering compaction.

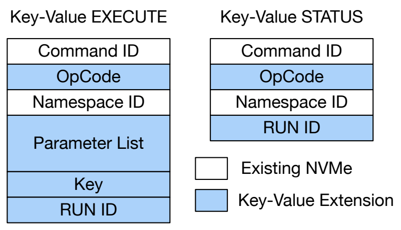


Figure 9. Key-value commands for data analytics extending NVMe protocols.

2) GET Operation: When a user application issues GET request, the corresponding request is passed to the *iLSM-SSD* through the Key-Value Device Driver. At this point, the page list includes kernel buffer pages allocated by the device driver, so that the value can be copied from the *iLSM-SSD* via DMA. When the GET request reaches the *iLSM-SSD*, it first examines the MemTable and the Immutable MemTable. If it finds the corresponding key, it reads the value using Value Log offset and copies the value to Host. Otherwise, it starts searching SSTable from the most recent SSTable of level 0. For each SSTable, first, check the key range from per-SSTable metadata. If requested key is in the key-range of SSTable, the bloom filter is loaded and examined. Finally, the index block is searched when result of bloom filter is positive. However, because the result of bloom filter can be false positive, the key may not be found in index block. In case of false positive, *iLSM-SSD* continues searching from next SSTable. Once the key is found the corresponding value is sent to host. Otherwise, it returns an error to host.

3) DELETE Operation: In case of DELETE operation, we simply issue PUT request with special value which indicates the key is deleted. The special value for DELETE operation is handled as same as PUT Operations in *iLSM-SSD*.

G. Data Analytics in iLSM-SSD

Near-data processing is a method of reducing data movement by performing data analytics at the location where the data is stored. Recent studies introduced near-data processing frameworks to process data in SSDs using internal computing resources such as CPU and memory [17], [18], [21]. These studies assume that near-data processing is performed on a block-by-block basis. That is, the user or application must tell the SSD which blocks to perform operations on. To do this, users and applications must find the data layout through the file system before performing near-data processing. This process still results in non-negligible file system interaction overhead and requires data movement between the host and the device. On the other hand, with *iLSM-SSD*, when performing near-data processing, the data layout can be obtained directly from the SSD, thus eliminating data movement. To this end, we design a key-value analytics framework to allow near-data processing in *iLSM-SSD*. In order to implement and use near-data processing in *iLSM-SSD*, it considers the followings: data analysis kernels to be served in the SSD, NVMe command definition for executing the data analytic kernels, and APIs to be used by data analytics logic.

1) Data Analytics Logic of User: In near-data processing, data analytics logic has to be provided by the user. For the user provided analytics logic, we store the binary of user-defined program as an executable key-value pair in *iLSM-SSD*. The executable key-value is stored in the same manner as any other key-value pairs using a PUT command of *iLSM-SSD*.

2) Storage Protocol for Key-Value Analytics: For the executable key-value, a protocol needs to be defined for the key-value data analytics. As shown in Figure 9, the EXECUTE and STATUS commands are defined by extending the NVMe protocols. The EXECUTE command has RUN ID, key, and parameter list field. The RUN ID is a unique identifier for each EXECUTE command and is used to find the status and return value of the command. The parameter list field holds a list of keys to be used as a parameter of the executable key-value by reusing the page list field in the PUT and GET commands. To query the current status of the EXECUTE command, we use the STATUS command with corresponding RUN ID. Possible states are WAITING, RUNNING, and EXITED, indicating when the EXECUTE command is waiting, running, or ended. When the EXECUTE command is terminated, it can return an 8 Byte single integer to the host that can be obtained by the STATUS command. Using the return value, the user can send back the result of the job as a single integer or a new key containing the result.

3) API for Key-Value Analytics: The data analytics logic may need to dynamically load keys that are not in the parameter key list during execution. For this end, we provide the same API as the NVMe key-value extension to the executable key-value. For example, consider a job is flipping all bits for every key in the parameter key list. The job can store the result of each key using the PUT command and create a meta key-value that contains the list of output keys. Then, the key of the meta key-value can be returned to the host.

IV. Evaluation

A. Evaluation Setup

To demonstrate the design of *iLSM-SSD*, we prototyped it on the Cosmos+ OpenSSD platform. Table I shows the details of the hardware we prototyped on. The Cosmos+ OpenSSD equips Xilinx Zynq-7000, which has an ARM Cortex-A9 processor with two cores, each running at 1GHz, FPGA and 1GB DDR3 DRAM. The FPGA operates SSD controller including NAND Flash controllers, NVMe controllers, and PCIe controllers. The page-level FTL and garbage collection are running at CPU. Though the Cosmos+

Table I
COSMOS+ OPENSSD PLATFORM SPECIFICATION.

SoC	Xilinx Zynz-7000
	ARM Cortex-A9 (up to 1000MHz)
	HYU Tiger 4 Controller (NVMe Controller in FPGA)
NAND Module	1TB module, NVDDR2
	4 Channel, 8 Way
	18048B Page (1664B Spare)
Interconnect	PCIe Gen2 8-lane
FTL	Page-level Mapping, On-demand GC

Table II
HOST MACHINE SPECIFICATION.

CPU	Intel Core i3-2100@3.1GHz 2C/4T
RAM	8GB
OS	Linux Kernel 4.19.43

OpenSSD has two cores, our *iLSM-SSD* software implementation utilizes only single core. The Cosmos+ OpenSSD is connected to the host machine via PCIe Gen2 8-lane with NVMe protocol. All experiments are run on the host machine as shown in Table II. The host machine has two cores and 4 threads with 8GB of DRAM, and runs Linux Kernel 4.19. For evaluation purposes, we implemented WiscKey based on RocksDB. We assume strong consistency scenario for key-value queries, which means that user's requests get responses only when changes are durable.

B. Results

For fair evaluation, the WiscKey was implemented in the RocksDB. And the performance of *iLSM-SSD* for PUT and GET operations are compared with the RocksDB performance.

1) PUT Performance: To evaluate the PUT performance, we generated sequential and random PUT workloads using 1 million of unique keys. We measured throughput and average latency by varying the value size.

Figure 10 shows average throughput and latency comparison results for WiscKey and *iLSM-SSD*. In the sequential PUT workload, all keys are sorted. There is no SSTable where key ranges overlap, even if compaction occurs. Therefore, compaction overhead is very negligible. As shown in Figure 10(a), *iLSM-SSD* shows twice as much throughput as WiscKey for all value sizes regardless of write patterns (sequential and random). In the case of WiscKey, compaction can be handled in the background using the host's spare CPU, but *iLSM-SSD* using single core has to deal with compaction too. This means that the *iLSM-SSD* is a better option than the WiscKey running on the host, even though *iLSM-SSD* has much less computing resources than the host.

We also observe that the throughput of both *iLSM-SSD* and WiscKey decreases as the value size increases. This is because time it takes to copy the key-value into device increases with value size. WiscKey should also call the *fdatasync* function to persist the value log for every PUT request. Unlike *iLSM-SSD*, WiscKey has to pay for not only the cost of data copying, but also the sync overhead of the file system. However, since the *iLSM-SSD* does not go through the file system, it only has the overhead for transmitting key-values.

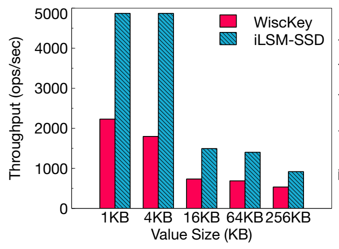
Figure10(b) shows the throughput comparison for random PUT workload. In the case of random PUT workload, the compaction cost increases because key ranges occur where SSTables overlap each other. *iLSM-SSD* showed similar throughput to sequential PUT for all value sizes. For WiscKey, compaction must read SSTables and write to them. Therefore, compaction and value log writes share I/O bandwidth between the host and the device. On the other hand, in case of *iLSM-SSD*, compaction consumes the internal bandwidth of SSD because compaction is performed inside of the device.

WiscKey showed lower throughput than sequential PUT when value size was 4 KB. However, *iLSM-SSD* showed similar throughput to sequential PUT for all value sizes. For WiscKey, compaction must read SSTables and write to them. Therefore, compaction and value log writes share I/O bandwidth between the host and the device. However, in case of *iLSM-SSD*, compaction consumes internal bandwidth of SSD because compaction is performed inside of the device. Therefore, even if compaction occurs, the PUT request uses all I/O bandwidth between the host and the device.

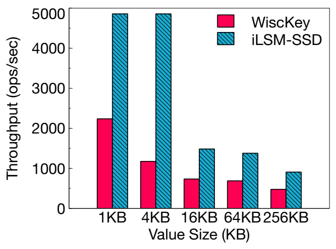
Figure 10(c)&(d) show average latency in sequential and random PUT workloads. In the case of WiscKey, compaction is performed using the background thread pool, and the foreground thread writes the value log. Thus, since multiple threads of the host can be used, the latency of WiscKey is relatively free from compaction overhead. However, SSD has a limited number of CPU cores. In the case of sequential PUT, compaction overhead is less than random PUT. Because *iLSM-SSD* performs compaction using a single core, the latency of PUT request includes compaction overhead. Nonetheless, in both sequential and random workloads, *iLSM-SSD* shows half-latency compared to WiscKey.

We also measured average latency of *fdatasync* in the WiscKey. *fdatasync* is called to persist Value Log entries for every PUT request. As shown in Figure 10(c)&(d), since WiscKey relies on the file system for persisting data, the average latency of PUT requests is mainly contributed by *fdatasync*. On the other hand, *iLSM-SSD* shows lower latency than *fdatasync* because PUT requests for *iLSM-SSD* do not involve the file system.

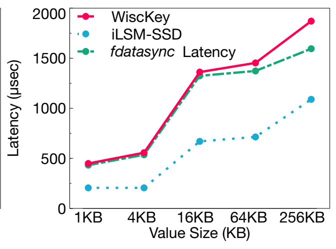
2) GET Performance: To evaluate the performance of a GET request, we generated a sequential and random GET workloads using 1 million unique keys. Figure 11(a)&(c) show throughput and latency comparisons for sequential GET workload. For sequential GET workload, WiscKey shows high throughput since it utilizes the iterator which



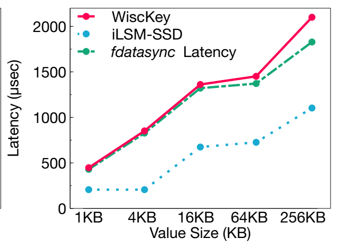
(a) Sequential PUT



(b) Random PUT

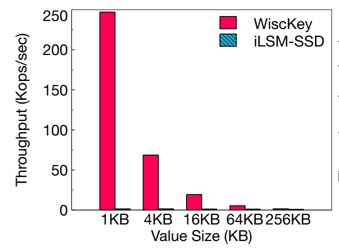


(c) Sequential PUT

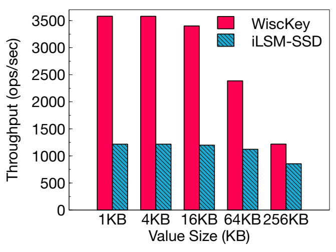


(d) Random PUT

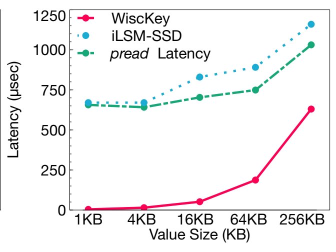
Figure 10. Performance comparison of *iLSM-SSD* and WiscKey for PUT workloads.



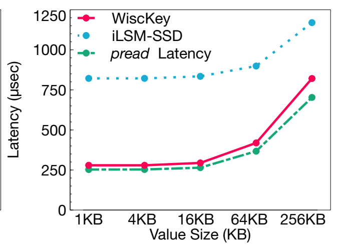
(a) Sequential GET



(b) Random GET



(c) Sequential GET



(d) Random GET

Figure 11. Performance comparison of *iLSM-SSD* and WiscKey for GET workloads.

is designed for the sequential access. Thus, when the next key is stored at the same SSTable, WiscKey can get the value by simply moving the iterator forward. Because consecutive keys do not require reading SSTable, the latency of *pread* is higher than GET latency of WiscKey. However, *iLSM-SSD* handles GET requests individually even if the access pattern is sequential. Therefore, the next key has to be searched through whole LSM-tree even if it is stored at the same SSTable.

Figure 11(b)&(d) show the performance comparison of random GET workload. In random GET Workload, the WiscKey can not utilize the iterator. Thus WiscKey and *iLSM-SSD* process each GET request separately even if GET requests are accessing consecutive keys. WiscKey shows better throughput than *iLSM-SSD*, but the throughput decreases as the value size increases. On the other hand, *iLSM-SSD* shows similar throughput for all value sizes. As shown in Figure 11(d), the latency of the GET request in the WiscKey is mostly contributed to *pread* latency. This is because WiscKey utilizes host's CPUs, which is more powerful than the CPU in SSD, for searching SSTables. The latency of *iLSM-SSD* shows higher than *pread*, because *iLSM-SSD* has to search SSTables serially since it uses a single core.

3) File System Layout Overhead: Near-data processing of block-based SSDs (NDP-SSD) must find out the physical layout of the data in user space and push it to the SSD. Time overhead can be divided into two time steps - (i) time to extract the phyaical data layout from the file system ($T_{FS\_Extract}$) and (ii) time to send it to

Table III
AVERAGE TIME (µSEC) OF EACH DELAY TO FIND OUT THE PHYSICAL LAYOUT OF THE DATA FOR NEAR-DATA PROCESSING.

	NDP-SSD		*iLSM-SSD*	
Delay	$T_{FS\_Extract}$	T_{Push_SSD}	$T_{MemTable}$	$T_{SSTable}$
(µsec)	4.25	522.00	1.14	663.62

SSD (T_{Push_SSD}). On the other hand, in *iLSM-SSD*, the layout of the data is managed by the LSM-tree. Therefore, only the LSM-tree search delay is added to the overhead for extracting the data layout in *iLSM-SSD*. More specifically, an LSM-tree search may be hit in the MemTable ($T_{MemTable}$), or it must search for the SSTable ($T_{SSTable}$).

We measured each time delay for NDP-SSD and *iLSM-SSD*. Table III shows the average time of each delay. WiscKey has to pay a time delay overhead of 526.25 µsec. On the other hand, *iLSM-SSD* only has to pay 1.14 µsec for the time delay in case of hit on the MemTable, or 664.76 µsec about 26% higher delay than NDP-SSD. However, the temporal locality of workloads and an appropriate amount of MemTables can reduce the miss rate in the MemTable.

V. Related Work

Key-Value Store for SSD: Various approaches have been proposed for building the key-value stores for SSD in host-side without the file system intervention [22]–[26]. NVMKV [22] proposed a hash-based key-value store to eradicate the write amplification by exploiting advance features of FTL such as sparse mapping. NVMKV uses the hash value of the key as an LBA and only maps the LBA range to NAND flash where key-value is stored. SkimpyStash [25] also proposed a hash-based approach to minimize DRAM footprint. SkimpyStash manages hash buckets using linear chaining while storing the head of the chain in DRAM and the rest of chain in SSD. SkimpyStash reduced latency for querying non-existing key by employing the per-bucket bloom filter. SILT [23] also employed the hash-based approach, but it also combines Log and Sorted Table. LOCS [26] implements an LSM-tree on open-channel SSD which exposes the internal geometry of SSD to the host. LOCS optimized compaction by locating SSTable considering NAND flash channel conflict. These approaches are optimized for memory footprint and do not have file system overhead because they are directly built on SSD. However, management of data stored in SSD such as linear probing in the hash table, data movement between layers or compaction still incur data movement between host and device. In *iLSM-SSD*, all management of LSM-tree such as compaction, GC and log cleaning is performed inside of SSD.

Implementing Key-Value Store in SSD: Several recent researches proposed the key-value store implementations inside of the SSD [27]–[30]. KAML [27] introduced hash-based key-value SSD to optimize the key-value performance. However, due to the hash table, it is difficult to perform operations requiring key order such as sequential scanning over keys. Also for the hash table, the load factor is crucial for the performance due to the hash collision. KVSSD [28] proposed the LSM-tree approach for Key-Value SSD and evaluated with the simulator. To mitigate write amplification problem caused by compaction, the remapping compaction is proposed that the resulting SSTable from compaction has a pointer to old SSTable to prevent rewriting same Key-Values. However, remapping compaction still leave invalidated key-value pairs. Thus, it will require to rewrite all SSTables which have been remapped before at some point later. LightStore [29] proposed the Key-Value SSD cluster. LightStore proposed LSM-tree based Key-Value SSD which is directly attached to the network and operates as a single server. On the other hand, *iLSM-SSD* attached as a local device to a host system. Kim et al. introduced the compound operation to hash-based Key-Value SSD to optimize key-value I/O performance [30].

Near-Data Processing Framework: SmartSSD [18], [19] enables users to launch customized tasks using CPU inside of SSD. Tasks running inside of SSD can be like map functions in Map-Reduce framework [18] or database query [19]. Biscuit [17] introduced near-data processing framework considering practical aspects. Biscuit defines the protocols for near-data processing and eases the development by supporting the full-featured standard library and modern C++ standards. BlueDBM [31] introduced flash array with near-data storage capability. In BlueDBM, storage nodes connected by dedicated storage network each other it can fully utilize the computing power of all nodes in the cluster. Summarizer [21] proposed the dynamic load balancing scheme for the near-data processing by implanting resource monitor in SSD. These studies propose and optimize the near-data processing frameworks on various dimensions such as load balancing between host and device, resource utilization in a storage array, and programmability. However, all these frameworks need to acquire the data layout on NAND flash via the file system [17], [31] or dedicated protocol [18], [19], [21]. In addition, when the key-value store is implemented inside of SSD like *iLSM-SSD*, near-data processing requests only need to inform which key to process to the device because the device already manages the data layout.

VI. Conclusion

The LSM-tree based key-value store operates on file systems and involves file system overhead. In particular, WiscKey, which is an approach to implement key-value separation, minimizes the write amplification problem caused by compaction of LSM-tree by writing value into Value Log and managing only key and value offset by LSM-tree. But WiscKey still could not completely hide the file system overhead. In this paper, we design and prototype an *iLSM-SSD* that runs WiscKey, an LSM-tree with key-value separation, in the SSD firmware. In particular, we propose scattered logging techniques to minimize the overhead of WiscKey's Value Log cleaning in a memory-constrained SSD environment. For fair evaluation, we implemented WiscKey in RocksDB and compare *iLSM-SSD* and the WiscKey. From extensive evaluations with synthetic benchmark workloads, we observed PUT performance was 1.6-4 times higher than WiscKey, but GET performance can be significantly lower. This low GET performance of *iLSM-SSD* can be improved if user-level caching or kernel-level caching is implemented.

Acknowledgments

We thank the reviewers for their constructive comments that have significantly improved the paper. This work was supported by the SK hynix research grant.

References

[1] New in Luminous: BlueStore. [Online]. Available: https://ceph.com/community/new-luminous-bluestore/

[2] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A distributed storage system for structured data," 2006.

[3] A. Lakshman and P. Malik, "Cassandra: a decentralized structured storage system," *ACM SIGOPS Operating Systems Review*, vol. 44, no. 2, pp. 35–40, 2010.

[4] Storage engine - Apache Cassandra 3.0. [Online]. Available: https://docs.datastax.com/en/cassandra/3.0/cassandra/dml/dmlManageOndisk.html

[5] A RocksDB storage engine with MySQL. [Online]. Available: https://myrocks.io

[6] Storage Engines - MongoDB Manual. [Online]. Available: https://docs.mongodb.com/manual/core/storage-engines/

[7] memcached - A Distributed Memory Object Caching System. [Online]. Available: https://memcached.org

[8] Redis. [Online]. Available: https://redis.io

[9] EVCache: Lowering Costs for a Low Latency Cache with RocksDB. [Online]. Available: https://www.percona.com/live/17/sessions/evcache-lowering-costs-low-latency-cache-rocksdb

[10] RocksDB. [Online]. Available: http://rocksdb.org

[11] LevelDB. [Online]. Available: https://github.com/google/leveldb

[12] Lightning Memory-Mapped Database Manager (LMDB). [Online]. Available: http://www.lmdb.tech/doc/

[13] Yahoo Sherpa. [Online]. Available: https://yahooeng.tumblr.com/post/120730204806/sherpa-scales-new-heights

[14] Apache Ignite. [Online]. Available: https://ignite.apache.org/use-cases/database/key-value-store.html

[15] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil, "The log-structured merge-tree (LSM-tree)," *Acta Inf.*, vol. 33, no. 4, pp. 351–385, Jun. 1996. [Online]. Available: http://dx.doi.org/10.1007/s002360050048

[16] L. Lu, T. S. Pillai, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "WiscKey: Separating keys from values in SSD-conscious storage," in *Proceedings of the 14th USENIX Conference on File and Storage Technologies*, ser. FAST '16, 2016.

[17] B. Gu, A. S. Yoon, D.-H. Bae, I. Jo, J. Lee, J. Yoon, J.-U. Kang, M. Kwon, C. Yoon, S. Cho *et al.*, "Biscuit: A framework for near-data processing of big data workloads," in *ACM SIGARCH Computer Architecture News*, vol. 44, no. 3. IEEE Press, 2016, pp. 153–165.

[18] Y. Kang, Y.-s. Kee, E. L. Miller, and C. Park, "Enabling cost-effective data processing with smart SSD," in *Proceedings of the 2013 IEEE 29th Symposium on Mass Storage Systems and Technologies*, ser. MSST '13. IEEE, 2013, pp. 1–12.

[19] J. Do, Y.-S. Kee, J. M. Patel, C. Park, K. Park, and D. J. DeWitt, "Query processing on smart SSDs: opportunities and challenges," in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. ACM, 2013, pp. 1221–1230.

[20] Cosmos+ OpenSSD Platform. [Online]. Available: http://openssd.io

[21] G. Koo, K. K. Matam, H. Narra, J. Li, H.-W. Tseng, S. Swanson, M. Annavaram *et al.*, "Summarizer: Trading communication with computing near storage," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. Micro '17. ACM, 2017, pp. 219–231.

[22] L. Marmol, S. Sundararaman, N. Talagala, and R. Rangaswami, "NVMKV: A scalable, lightweight, FTL-aware key-value store," in *Proceedings of the 2015 USENIX Annual Technical Conference*, ser. ATC '15, 2015, pp. 207–219.

[23] H. Lim, B. Fan, D. G. Andersen, and M. Kaminsky, "SILT: A memory-efficient, high-performance key-value store," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, ser. SOSP '11. ACM, 2011, pp. 1–13.

[24] B. Debnath, S. Sengupta, and J. Li, "FlashStore: High throughput persistent key-value store," *Proceedings of the VLDB Endowment*, vol. 3, no. 1-2, pp. 1414–1425, 2010.

[25] B. Debnath, S. Sengupta, and J.Li, "SkimpyStash: RAM space skimpy key-value store on flash-based storage," in *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*. ACM, 2011, pp. 25–36.

[26] P. Wang, G. Sun, S. Jiang, J. Ouyang, S. Lin, C. Zhang, and J. Cong, "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*, ser. Eurosys '14. ACM, 2014, p. 16.

[27] Y. Jin, H.-W. Tseng, Y. Papakonstantinou, and S. Swanson, "KAML: A flexible, high-performance key-value SSD," in *Proceedings of the 2017 IEEE International Symposium on High Performance Computer Architecture*, ser. HPCA '17. IEEE, 2017, pp. 373–384.

[28] S.-M. Wu, K.-H. Lin, and L.-P. Chang, "KVSSD: Close integration of LSM trees and flash translation layer for write-efficient kv store," in *Proceedings of the 2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2018, pp. 563–568.

[29] C. Chung, J. Koo, J. Im, S. Lee *et al.*, "LightStore: Software-defined network-attached key-value drives," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2019, pp. 939–953.

[30] S.-H. Kim, J. Kim, K. Jeong, and J.-S. Kim, "Transaction support using compound commands in key-value SSDs," in *11th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 19)*. USENIX Association, 2019.

[31] S.-W. Jun, M. Liu, S. Lee, J. Hicks, J. Ankcorn, M. King, S. Xu *et al.*, "BlueDBM: An appliance for big data analytics," in *Proceedings of the 2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture*, ser. ISCA '15. IEEE, 2015, pp. 1–13.