LevelDB-Raw: Eliminating File System Overhead for Optimizing Performance of LevelDB Engine

Hak-Su Lim and Jin-Soo Kim

*College of Info. & Comm. Engineering, Sungkyunkwan University, Korea

{haksu.lim, jinsookim}@skku.edu

***Abstract*— Key-value store, a kind of NoSQL database, is data storage system for storing and retrieving key-value pairs using unique keys. In this paper, we present LevelDB-Raw, which improves LevelDB, one of the most well-known key-value store engines. LevelDB-Raw opens a raw block device file directly, bypassing the existing file system layer. Instead, it implements a simple, light-weight user-level file system to eliminate file system overhead such as redundant journaling and metadata management. We demonstrate the performance improvement of LevelDB-Raw using ForestDB-Benchmark and YCSB workloads. The results indicate that LevelDB-Raw increases the overall performance by 1.16x-3.45x in HDD and 1.05x-2.26x in SSD for updating database.**

***Keywords*— Key-value store, LevelDB, NoSQL**

I. INTRODUCTION

As online services, such as web indexing, SNS and cloud computing, are getting bigger and bigger, quick and efficient data processing methodology is required to provide data services in server environments with limited resources. This trend results in a radical shift toward NoSQL databases from conventional relational databases. Compared to relational database systems, NoSQL systems are faster and easier to use with unstructured data, thanks to their simpler structures.

However, tremendous time and money are still required to cover massive client requests due to the increasing amount of data to be processed. As a result, considerable efforts have been made to optimize the performance of NoSQL systems.

In this paper, we focus on a key-value store called LevelDB [1]. LevelDB is one of the widely-used key-value stores designed by Google. Our goal is to develop the improved version of LevelDB called LevelDB-Raw which works on a raw block device directly, bypassing the existing file system layer. Instead, LevelDB-Raw internally has a simple, light-weight user-level file system to eliminate file system overhead such as redundant journaling and metadata management.

The remainder of the paper is organized as follows. Section II briefly outlines the related work. Section III describes the architecture of LevelDB-Raw. In section IV, we explain implementation details of LevelDB-Raw. Section V provides experimental results compared to the original LevelDB. We conclude the paper in Section VI.

II. RELATED WORK

Most key-value stores use LSM-tree [2] for indexing key-value pairs. LSM-tree was originally introduced for HDDs (Hard Disk Drives) to overcome their significantly poor random write performance by writing data sequentially to the storage. However, key-value stores based on LSM-tree still suffer from performance degradation on HDDs when retrieving a key-value pair as it requires random reads. In this regard, recent SSDs (Solid State Drives) have potential to improve the performance of LSM-tree since SSDs show relatively faster random read performance than HDDs by servicing multiple read requests concurrently using the device internal parallelism.

Leveraging this SSD characteristic, LOCS [3] attempts to optimize LevelDB by using open-channel SSD. To take advantage of internal parallelism of the SSD, LOCS uses multiple threads to write data concurrently from memtable of LevelDB to the SSD. Also, LOCS proposes load balancing of write operations and delayed delete operation in the I/O request scheduling layer.

WiscKey [4] finds that, as the LSM-tree size grows, write and read amplification are getting bigger due to LevelDB's compaction operation. This drastically decreases the performance of LSM-tree-based key-value stores and reduces device lifetime. To address this issue, WiscKey separates keys from values and makes the value of LSM-tree point to the LSM-tree log without storing the real value of the corresponding key in LSM-tree. In this way, WiscKey reduces the size of LSM-tree considerably, resulting in better performance compared to the original LevelDB. Furthermore, WiscKey takes advantage of multi-threading for random reads to exploit abundant internal parallelism of SSD devices.

III. LEVELDB-RAW

To deliver high performance, LevelDB-Raw maintains a simple, light-weight file system inside of the LevelDB library as shown in Figure 1, bypassing the kernel file system such as ext4 [5] and f2fs [6]. The LevelDB-Raw file system includes only the essential and minimal data structure and functionality that are required to run the LevelDB only. Thus, LevelDB-Raw generates much smaller I/O traffic than LevelDB running on top of the kernel file system.

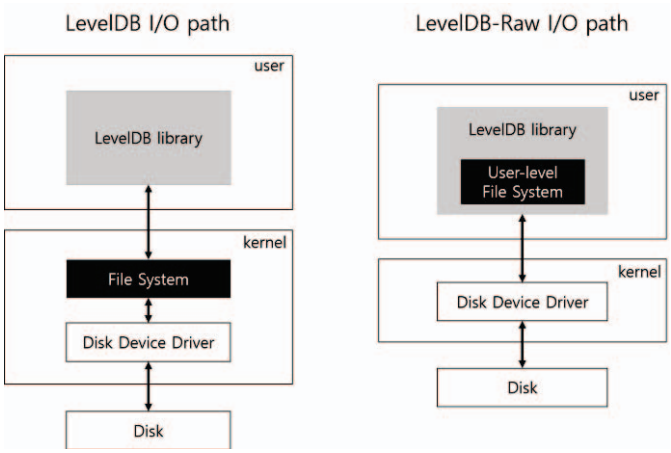

Figure 1. LevelDB & LevelDB-Raw I/O path

Note that the LevelDB stores the incoming key-value pair to the log file first to ensure consistency against sudden power outage or hardware/software failures. Even though LevelDB has its own application-level crash recovery mechanism, the traditional file system performs redundant journaling or checkpointing at the file system layer again, resulting in "Journaling of journal (JoJ) [7]" problem. LevelDB-Raw eliminates these overheads by passing the responsibility of crash recovery to LevelDB.

We have modified only the LevelDB library in the user space to develop LevelDB-Raw. Therefore, LevelDB-Raw is very portable and can be run on any Linux kernel versions.

IV. IMPLEMENTATION

A. User-level File System for LevelDB-Raw

There are six kinds of files generated by LevelDB as shown in Table 1. Most dominant file types are both ldb (*.ldb) files and log (*.log) files. The ldb file contains a set of key-value pairs while the log file is used for maintaining consistency of LSM-tree. The size of each ldb file is about 2MB and the size of a log file is less than 2MB. LevelDB-Raw exploits these characteristics to simplify the file system design. Since the file size is almost same, our user-level file system allocates a fixed size storage space to each file, which is slightly larger than 2MB. To keep track of file allocation information, only file names and their start block addresses are maintained by a hash table. For example, when LevelDB needs to create a file, our user-level file system searches for an empty space in the storage, assigns the space to the target file, and records the file

TABLE 1. FILES GENERATED BY LEVELDB

*.ldb	Set of key-value pairs
*.log	Log file for consistency of LSM-tree
MANIFEST-*	Core metadata file of LevelDB
CURRENT	Pointing current MANIFEST file
LOCK	Lock file for opening and closing DB
LOG	LevelDB trace log

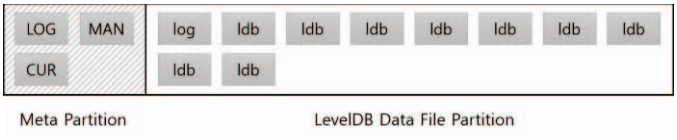

Figure 2. LevelDB-Raw Storage Layout

name and the associated block address to the hash table. To make the file system consistent, LevelDB-Raw synchronizes this hash table when LevelDB generates fsync() operation to finish writing files.

Other types of files include MANIFEST, LOG, CURRENT and LOCK files, which are used to store certain metadata for LevelDB. Since they generate a negligible amount of I/O traffic and can be accessed by other utilities for maintenance purpose, we simply allocate them in the existing kernel-level file system. As shown in Figure 2, we divide the whole disk into "meta partition" and "data file partition". We format the meta partition using a kernel-level file system and use it for storing LevelDB metadata files. On the other hand, LevelDB-Raw manages the data file partition directly at the user space and uses it for allocating ldb and log files. One limitation of this design is that users cannot access ldb and log files using the system utilities (such as /bin/cat, etc.) based on the kernel-level file system. However, it is very uncommon for users to manipulate those files directly because they contain complicated structure for users to get any information about the stored key-value data.

B. Padding Small Writes

LevelDB-Raw opens a raw block device directly in order to bypass the existing file system. When LevelDB-Raw creates a new file and writes data to it, LevelDB-Raw needs to (1) search for an empty space, (2) update the hash table, (3) and finally write data to the given space. We find that writing data to the block device directly has an unexpected performance penalty especially when the data size is less than the kernel block size (e.g. 4KB in Linux). When we write a small amount of data, say 100 bytes, to a certain block in the block device, the kernel attempts to temporarily buffer the data in the page cache. Even though this is a new write to an empty block, the kernel is not aware of the fact. Thus, the kernel

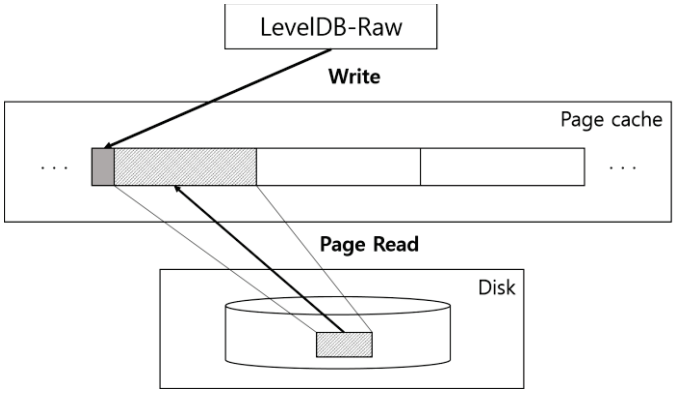

Figure 3. Partial Write

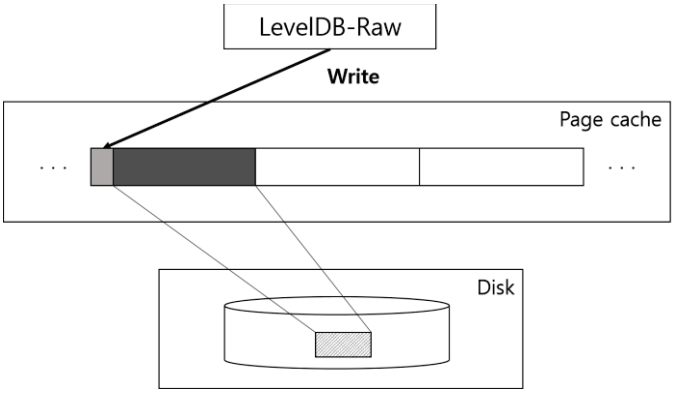

Figure 4. Padding Write

allocates a page in the page cache and fills the page with the old data read from the storage. And then, the first 100 bytes in the page are overwritten according to the user's request as shown in Figure 3; we refer to this as the *partial write*. This means that whenever an empty block is written with the partial data, an unnecessary block read operation from the disk is performed which is very expensive.

To address this problem, LevelDB-Raw performs *padding writes* as shown in Figure 4. To eliminate the expensive block read operation from the disk, LevelDB-Raw always writes 4KB of data to an empty block by padding the remaining area with the stale data. In this case, the kernel simply copies the data to a page allocated in the page cache, without issuing any block read operation to the disk. A drawback of this approach is that the memory copy overhead is slightly increased due to the padded data. However, it is relatively cheaper and negligible compared to the overhead of reading a block from the disk.

V. Evaluation

A. Evaluation Methodology

We use ForestDB-Benchmark and YCSB [8] (Yahoo! Cloud Serving Benchmark) workloads to evaluate the performance of LevelDB-Raw. ForestDB-Benchmark updates and reads a number of key-value pairs after loading one million key-value sets whose size is 512 bytes. When running ForestDB-Benchmark, we can specify the *batch* size. The batch size of 1024 means that LevelDB calls the fsync() system call after every 1024 operations. In the experiments, we have used the batch size of 1 and 1024.

TABLE 2. YCSB Workload Specifications

Workload-a	Read : Update = 50 : 50
Workload-b	Read : Update = 95 : 5
Workload-c	Read : Update = 100 : 0
Workload-d	Read : Insert = 95 : 5
Workload-e	Scan : Update = 95 : 5

The experiments with YCSB workloads are conducted after loading one hundred thousand 1KB-sized key-value pairs.

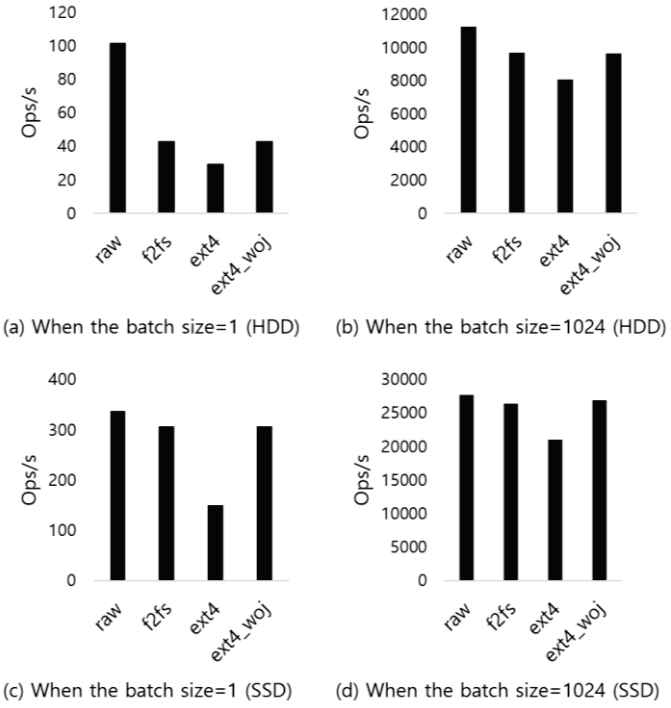

Figure 5. ForestDB-Benchmark: Update Performance

Table 2 shows the detailed specifications of YCSB workloads used in this paper. YCSB workloads perform four kinds of operations: read, update, scan, and insert. The read operation searches for a key that is randomly generated. The update operation also overwrites values corresponding to the randomly generated keys. The insert operation literally adds new key-value pairs. Lastly, the scan operation sequentially reads key-value pairs.

We compare the performance of LevelDB-Raw with that of LevelDB when it runs on two file systems, ext4 and f2fs. Table 3 summarizes the experimental setup.

TABLE 3. Experiment Environments

CPU	Intel® Core™ i7-4790 3.6GHz 8cores
CPU Cache	L1 : 256KB L2 : 1MB L3 : 8MB
RAM	Samsung DDR3 4GB * 2
OS	Linux Ubuntu server 4.4.19
HDD	Seagate ST1000DM0003-1ER1 1TB
SSD	Samsung 850 Pro 256GB

A. ForestDB-benchmark Results

As shown in Figure 5, our experimental results show that LevelDB-Raw outperforms the original LevelDB both with the batch size of 1 and 1024. The main reason of the improved performance is due to the reduced amount of I/Os.

An interesting aspect of this experiment is the result of ext4_woj. Ext4_woj represents the case where LevelDB is executed on the ext4 file system with journaling turned off. Figure 5 shots that the performance of ext4 without journaling

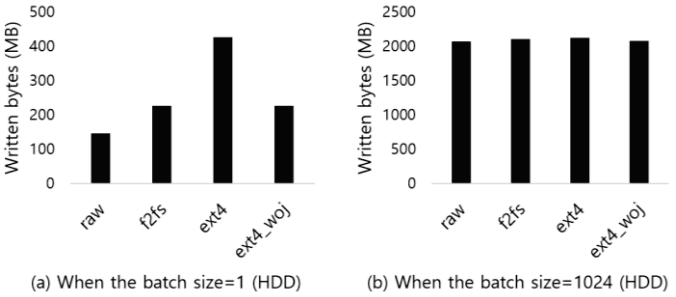

Figure 6. ForestDB-Benchmark: Written Bytes

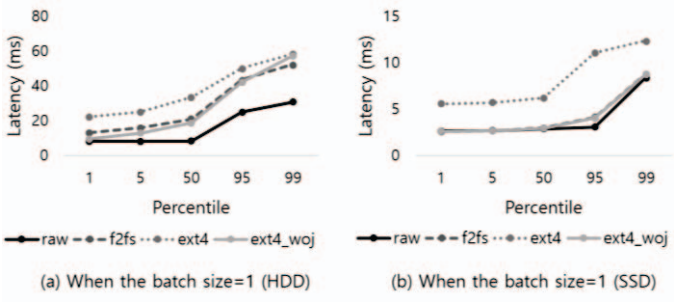

Figure 7. ForestDB-Benchmark: Update Latency Distribution

is almost identical to that of f2fs. This is because the amount of I/O generated by the ext4 file system without journaling nearly equals to that by the f2fs file system.

Furthermore, we can notice that there is more performance improvement when the HDD is used as a storage device than the case with the SSD. This is due to the random write requests generated by fsync() system call that file system generates for updating metadata. Random writes are critical for I/O performance in the HDD since those require disk seeks. On the other hand, random writes in the SSD do not cause a large slowdown due to the internal parallelism inside the device.

In addition, LevelDB-Raw leads to a larger performance improvement when the batch size is small. This is also because random write requests generated by file system. The fsync() system call generated by file system is more frequently called when the batch size is small, which generates more metadata writes to ensure the file system consistency. Thus, the update performance of existing file system gets worse as the batch size decreases.

Figure 6 shows the amount of written data generated by each file system when the same number of key-value pairs are written. In Figure 6(a), we can see that the amount of written data to the storage device is increasing in order of LevelDB-Raw, f2fs and ext4. This means LevelDB-Raw generates the smallest amount of file system metadata. However, Figure 6(b) depicts that there is no notable difference in the amount of written bytes when the batch size is increased to 1024. This is because the amount of file system metadata generated by the fsync() system call is negligible when the batch size is large. Also, similar to the result in Figure 5, we can observe that the

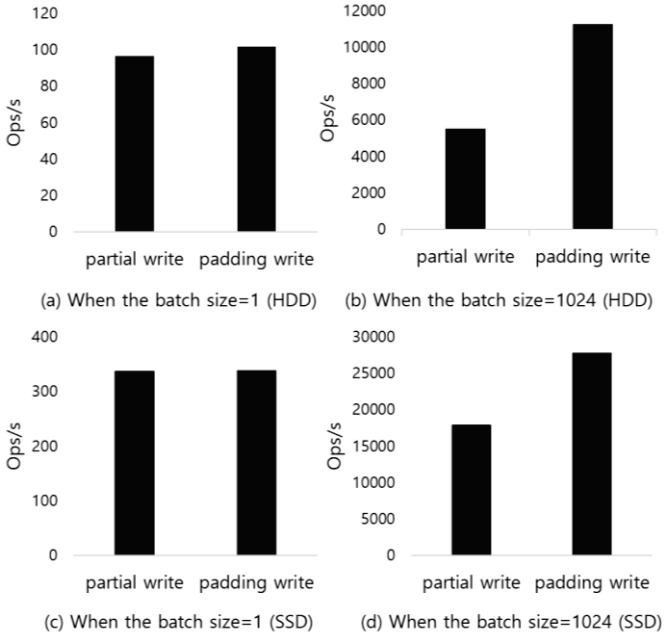

Figure 8. ForestDB-Benchmark: The Effect of Padding Writes

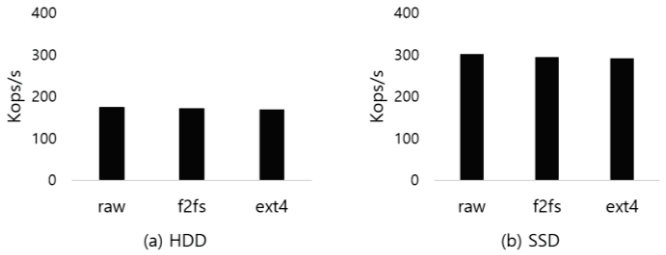

Figure 9. ForestDB-Benchmark: Lookup Performance

amount of written bytes in the ext4 file system without journaling (ext4_woj) is comparable to that in the f2fs file system.

Figure 7 illustrates the 1st, 5th, 50th, 95th, and 99th percentile latency of update requests. In HDD, LevelDB-Raw shows the lowest latency while ext4 with journaling exhibits the highest latency. LevelDB-Raw and f2fs show similar latencies in SSD, possibly because the metadata size generated by the fsync() system call is almost same in both cases.

Figure 8 compares the effect of padding writes on the overall performance. If padding write is not used, partial write occurs when the corresponding page does not exist in the page cache. In this experiment, we have used 32-byte keys and 512-byte values, so partial write is performed in almost every 7 write operations when the batch size is set to 1. Thus, performance improvement is small in this case. On the other hand, when the batch size is set to 1024, every write request generate partial write, because LevelDB generates write requests as slightly less than 32KB size which is maximum size of LevelDB write request and this request makes partial write. Therefore the batch update has a larger performance improvement than the update without batch. Performance is increased by 1.05x-2.03x in HDD and 1.004x-1.54x in SSD.

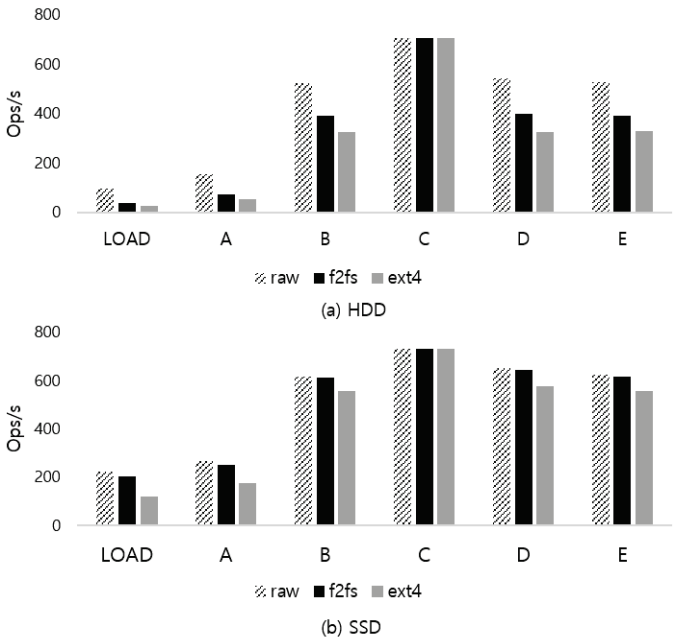

Figure 10. YCSB Performance

Finally, Figure 9 illustrates the lookup performance of LevelDB. Because ForestDB-Benchmark uses the zipf distribution when generating keys, almost all of the key-value pairs are cached in memory, so that the disk traffic is significantly reduced. Therefore, the lookup performance is almost identical to each other.

Overall, ForestDB-benchmark results show that LevelDB-Raw increases the overall throughput by 1.16x-3.45x in HDD and 1.05x-2.26x in SSD for updating database.

B. YCSB Results

In case of YCSB, LevelDB-Raw outperforms the original LevelDB in almost all of the workloads as shown in Figure 10. Unlike ForestDB-Benchmark, YCSB has a mix of read and write operations. LevelDB-Raw performs better in case of update operations. Hence, the higher ratio of write operation the workload has, the better performance improvement the result shows. We can see that the results show the large performance improvement in the case where the key-value pairs are initially loaded (LOAD in Figure 10) and in workload A which has the write ratio of 50 percent. As in the results of ForestDB-Benchmark, the performance improvement has been larger on the HDD than on the SSD.

VI. CONCLUSION

As the amount of data generated on the Internet is growing at an exponential rate, key-value stores play an important role in scalable data processing services. The proposed LevelDB-Raw aims at improving the performance of LevelDB by inserting a simple, light-weight user-level file system between LevelDB and the block device. Through extensive evaluations, we show that this approach is effective in eliminating file system overhead such as redundant journaling and extra I/O traffic caused by metadata writes. We believe that the similar approach can be also helpful for other key-value stores and data-intensive applications.

ACKNOWLEDGMENT

This work was supported by the National Research Foundation of Korea (NRF) grant founded by the Korea Government (MSIP) (No. NRF-2016R1A2A1A05005494).

REFERENCES

[1] J. Dean and S. Ghemawat, "LevelDB," http://code.google.com/p/leveldb, 2011.

[2] P. O'Neil, E. Cheng, D. Gawlick and E. O'Neil, "The log-structured merge-tree (LSM-tree)," Acta Informatica, vol. 33, no. 4, pp.351-385, 1996

[3] 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," the Ninth European Conference on Computer Systems, pp. 16, 2014

[4] L. Lu, T. S. Pillai, A. C. Arpaci-Dusseau and R. H. Arpaci-Dusseau, "WiscKey: separating keys from values in SSD-conscious storage," 14th USENIX Conference on File and Storage Technologies, pp. 133-148, 2016

[5] A. Mathur, M. Cao, S. Bhattacharya, A. Dilger, A. Tomas and L. Vivier, "The new ext4 filesystem: current status and future plans," Linux symposium, vol. 2, pp. 21-33, June. 2007

[6] C. Lee, D. Sim, J. Hwang and S. Cho, "F2FS: A new file system for flash storage," 13th USENIX Conference on File and Storage Technologies, pp. 273-286, 2015

[7] S. Jeong, K. Lee, S. Lee, S. Son and Y. Won, "I/O Stack Optimization for Smartphones," 2013 USENIX Annual Technical Conference, pp. 309-320, 2013

[8] B. F. Cooper, A. Silberstein, E.Tam, R. Ramakrishnan and R. Sears, "Benchmarking cloud serving systems with YCSB," 1st ACM symposium on Cloud computing, pp. 143-154, 2010

Hak-Su Lim received his B.S., degrees in Computer Engineering from Dongguk University, Republic of Korea, in 2013.
He also worked at modem development department of Samsung Electronics from 2013 to 2014.
He is currently a MS candidate in Computer Science at Sungkyunkwan University. His research interests include key-value store, storage system, embeded system, operating system.

Jin-Soo Kim received his B.S., M.S., and Ph.D. degrees in Computer Engineering from Seoul National University, Republic of Korea, in 1991, 1993, and 1999, respectively.
He is currently a professor at Sungkyunkwan University. Before joining Sungkyunkwan University, he was an associate professor at Korea Advanced Institute of Science and Technology (KAIST) from 2002 to 2008. He was also with the Electronics and Telecommunications Research Institute (ETRI) from 1999 to 2002 as a senior member of the research staff, and with the IBM T.J. Watson Research Center as an academic visitor from 1998 to 1999. His research interests include embedded systems, storage systems, and operating systems.