LSM-KVD: LSM-tree based Key-Value SSD

(Abstract) This is a template of Extended Abstract for HumanTech Paper Award. The recommended volume is 2 pages with 2-column format. Titles do not exceed two lines and abstracts do not exceed 15 lines.
papers should be written in Times New Roman font with the font size of 20pt in bold for the title, 10pt in bold for abstracts, 11pt in bold for the titles within text, 10pt for the text, 9pt in bold for the titles of figures and tables and 9pt for the references. For the fairness of the review, Name, major, the school/university name, the school/university logo, and teacher/professors name of author should not be included in the abstract and paper.

1. INTRODUCTION

Key-Value store (KVS) has become a necessary infrastructure for many applications, such as caching, web indexing, and storage systems. Owing to its popularity and huge impact on application performance, serious efforts have been made to optimize KVS in various ways. Recently, offloading most of the key-value (KV) functionality onto a storage device has been suggested, the so-called KV-SSD. By making the storage hardware directly serve KV requests via a driver, KV-SSDs enable clients to bypass the deep software stack in the host. For example, by performing most commonly used operations of KV databases, such as RocksDB, in KV-SSDs, we not only improve I/O latency and throughput of the operations but also reduce host resources *i.e.,*CPU cycles, Memory.

The existing ideas using hash algorithm for managing KV pairs because it has a simple architecture. However, if the DRAM size is not large enough to hold all the hash entries, parts of the hash table must be stored in flash. This inevitably involves expensive flash accesses to look for a key when it is not found in the DRAM-resident hash table. Even worse, if a hash collision occurs then multiple flash accesses may be required, resulting in long tail latency and drop in throughput.

An alternative to hashing is a *log-structured merge tree* (LSM-tree) [5] which can be widely used for KV-store. However, it has not only bad average read performance, but also addtional overheads for the writing compaction. Since one of its processes is sorting [4] keys, CPU cycles can become a huge burden on embedded CPUs in SSD controllers. Also extra I/O operations to keep KV pairs sorted further deteriorate I/O throughput. Due to these drawbacks, LSM-tree has not been considered an attractive solution for KV-SSDs.

In this paper, by optimizing the LSM-tree algorithm and tightly integrating it with an SSD controller, LSM-tree-based KV-SSDs can outperform hash-based designs. Our solution is based on three ideas. First, pin all KV indices of the top levels of the tree to DRAM to speedup reads. This level pinning not only guarantees the worst-case read latency, but improves average latency by reducing flash look-ups for retrieving KV indices. Second, run sorting tasks using HW accelerators, in parallel with I/O tasks to eliminate CPU overheads for compaction. Third, combine level-pinning with key-value separation [4] to reduce I/O overheads for compaction. Collectively these techniques remove the compaction bottleneck and provide high write-throughput.

Additionally, due to the first idea, we can extend the design space of LSM-tree-based KV-SSD by using trade-off between DRAM requirements and the performance. We believe that our techniques can also be applied profitably to general LSM-tree implementations, which may not have the same resource constraints as KV-SSDs do.

2. LSM-KVD

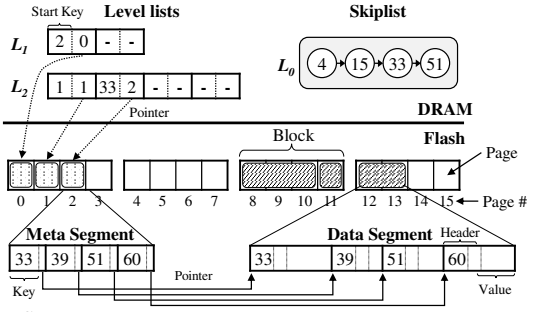

Fig. 1. LSMKVD Layout

2.1. Design of LSM-KVD

LSM-KVD has four types of data structures to implement the LSM-tree algorithm (Figure 1): a *skiplist* and *level lists*, which all reside in DRAM, and *meta segments* and *data segments*, which all reside in flash. The skiplist corresponds to L_0 in the LSM-tree algorithm, and holds KV objects as large as enough to fully utilize the parallelism of a NAND chip array. Each skiplist entry has four fields: <key size, key, value size, value>, and the skiplist is kept sorted by the keys. When the skiplist is full, it flush all KV pairs in skiplist into data segments and make a new meta segment for the flushed KV objects. The new meta segment is also written into the flash, and inserted into L_1 *level list*.

level list keeps track of meta segments at every level in the flash, Each level list is an array of pointers(4B each) to meta segments belonging to the level. Each array entry also contains the start key (a variable size from 16B to 128B) of the meta segment to facilitate searching. Figure 1 illustrates examples of how the level lists point to meta segments which, in turn, point to actual values. when a specific level(L_n) list is full, the level list compacts with below level(L_{n+1}) list and becomes a new lower level(L_{n+1}).

2.2. DRAM Requirements Comparison

Let's estimate DRAM requirements of LSM-KVD and Hash-based KV-SSD. We assume that an SSD capacity is 4 TB and each meta segment is 32 KB. First consider the case where the average sizes of keys and values are 32 B and 1 KB, respectively [1]. In this case only 162 MB DRAM is needed to hold all the level lists. For the worst-case scenario, where all the objects have the maximum key size of 128 B, the level lists would require about 2.1 GB DRAM. This can still easily fit in 4 GB DRAM that 4 TB SSD has for mapping.

However, in case of Hash-based one, it needed at least 144GB DRAM in 32B keys and 1KB values setting. To reduce DRAM size, some use signature instead of key [3] by hashing its key. Although it reduces the DRAM requirements to 24GB which is calculated when its each signature size is 2B, it is too huge to fit caching all the mapping. It means that even using the signature design it should store some part of the mapping in flash chips. And it also occur another type of collision *i.e.,*the different two keys can have a same signature.

2.3. Optimizing Technics

Optimizing Read Latency: The traditional LSM-tree uses bloom filter for optimizing read. But LSM-KVD adopts more aggressive strategy: *level pinning*. If the LSM-tree has n levels, LSM-KVD keeps meta segments for top-k levels ($k \leq n-1$) in DRAM. To process a `GET` request, it first searches for a key in top-k levels in DRAM. Only when a key is not found there, it lookups the rest of levels resident in the flash.

Optimizing Compaction: To keep each level sorted, LSM-KVD performs compaction which involves extra I/Os and CPU cycles. By using *level pinning* and seperating keys from values [4], it can reduce I/O traffics in compaction phase. However, The sorting overhead of compaction is pretty huge on embedded system. We address this problem by offloading sorting tasks to a special hardware accelerator in the SSD controller.

2.4. Memory and Performance Trade-offs

Thanks to *level pinning*, We extend LSM-tree design trade-off using not only the height but also number of pinned levels. The original cost formation of LSM-tree's average write throughput and worst look-up cost are $O((n-1)\cdot\frac{R^{1/(n-1)}}{B})$ and $O(n-1)$ where R is the number of meta segments in as SSD, and B is the average number of keys in a meta segment.

But when using *level pinning*, the write costs and look-up costs are change to $O((n-k-1)\cdot\frac{R^{1/(n-1)}}{B})$ and $O(n-k-1)$ respectively. Also the DRAM requirements of *level pinning* can be modeled $\sum_{i=1}^{k}(R^{1/(n-1)})^i \cdot M$ where M is the size of a meta segment. Thus, by using two parameters, We can more easily design LSM-KVD to fit DRAM size and performance requirements.

3. EXPERIMENTS

3.1. Experimental Setup

We implemented the LSM-KVD software and hardware accelerator on our FPGA-based SSD platform that has a quad-core ARM Cortex-A53 CPU running at 1.2 GHz with 16 GB DRAM. This hardware specification is almost equivalent to those of recent SSDs.

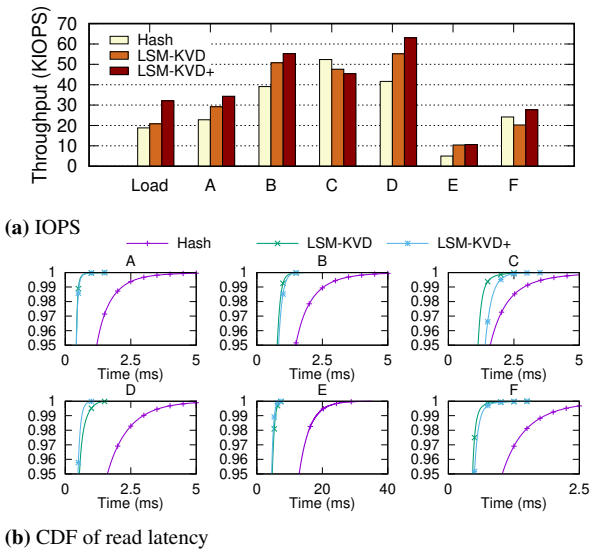

Fig. 2. Throughput comparison of Hash-based KV-SSD and LSM-KVD

For realistic experiments, We implemented the hash-based KVSSD using 8 bits signature. It makes more entries caching. And LSM-KVD is pinning top 3 levels among the total 5 levels. To confirm effect of the HW-accelerator, We tested two diffrent versions of LSM-KVD. One is `LSM-KVD`, another is `LSM-KVD+` which adopt the HW-accelerator .

3.2. YCSB

We measured IOPS and read latency of the hash-based KV-SSDs and LSM-KVD while running YCSB. YCSB has variable workloads with read,write ratio and workload pattern. Please see [2] for workloads details.

IOPS: Figure 2 (a) shows IOPS result of YCSB. LSM-KVD outperformed the hash-based ones, providing 47% higher IOPS, on average. Particularly, for YCSB-E with range queries, LSM-KVD showed 71∼111% higher IOPS thanks to LSM-tree's sorted nature.

Read Latency: Figure 2 (b) shows CDF graphs of read response times of all three types of KVSSD from 95% to 99.99%. `Hash` was suffer collision on all workloads. However, By using level pinning LSM-KVD can guarantee that `GET` can be done in two flash lookups. Thus, LSM-KVD beat `Hash` on all workloads.

REFERENCES

[1] Atikoglu, B., Xu, Y., Frachtenberg, E., Jiang, S., and Paleczny, M. Workload Analysis of a Large-scale Key-value Store. In *Proceedings of the ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems* (2012), pp. 53–64.

[2] Cooper, B. F., Silberstein, A., Tam, E., Ramakrishnan, R., and Sears, R. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the ACM symposium on Cloud computing* (2010), pp. 143–154.

[3] JUN, S.-W., LIU, M., LEE, S., HICKS, J., ANKCORN, J., KING, M., XU, S., AND ARVIND. BlueDBM: An Appliance for Big Data Analytics. In *Proceedings of the Annual International Symposium on Computer Architecture* (2015), pp. 1–13.

[4] LU, L., PILLAI, T. S., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. WiscKey: Separating Keys from Values in SSD-conscious Storage. In *Proceedings of the USENIX Conference on File and Storage Technologies* (2016), pp. 133–148.

[5] O'NEIL, P., CHENG, E., GAWLICK, D., AND O'NEIL, E. The Log-structured Merge-tree (LSM-tree). *Acta Informatica 33*, 4 (1996), 351–385.