KVSSD: Close Integration of LSM Trees and Flash Translation Layer for Write-Efficient KV Store

Sung-Ming Wu, Kai-Hsiang Lin, and Li-Pin Chang
Department of Computer Science, National Chiao-Tung University
Hsinchu, Taiwan, R.O.C.
tbches.cs03g@nctu.edu.tw, k99182003@gmail.com, lpchang@cs.nctu.edu.tw

***Abstract*—Log-Structured-Merge (LSM) trees are a write-optimized data structure for lightweight, high-performance Key-Value (KV) store. Solid State Disks (SSDs) provide acceleration of KV operations on LSM trees. However, this hierarchical design involves multiple software layers, including the LSM tree, host file system, and Flash Translation Layer (FTL), causing *cascading write amplifications*. We propose KVSSD, a close integration of LSM trees and the FTL, to manage write amplifications from different layers. KVSSD exploits the FTL mapping mechanism to implement copy-free compaction of LSM trees, and it enables direct data allocation in flash memory for efficient garbage collection. In our experiments, compared to the hierarchical design, our KVSSD reduced the write amplification by 88% and improved the throughput by 347%.**

I. INTRODUCTION

Relational database management systems (RDBMS) provide rich features such as transaction consistency and SQL queries. However, these features add high complexity to data processing and may degrade the performance of data-intensive applications. Recently, persistent Key-Value (KV) store is becoming an important component in the infrastructure of large-scale data centers. Compared to RDBMS, KV store offers a small set of operations for data store and retrieval. Because KV store is highly efficient and scalable, it has been widely used in in-line storage deduplication [1], e-commerce [2], and social networks [3].

KV store contains a large set of KV pairs, which are indexed by their key field. It can be implemented using hash tables [4], [5], [6], B-tree variants [7], or Log-Structured Merge (LSM) trees [8], [9], [10]. Among these options, LSM trees are highly write-optimized: They flush buffered KV pairs to the storage in terms of large Sorted-String Tables (SSTables) through out-of-place sequential logging. To further boost the throughput of KV operations, recent studies suggest to replace hard drives with NAND-flash-based SSDs [1], [9]. By this design, KV store involves multiple software layers, including the LSM tree, the host file system, and the firmware inside of SSDs. The layered design offers excellent modularity, but it risks *cascading write amplifications*, which negatively impact on the KV performance and seriously degrade the SSD lifespan. Specifically, the *write amplification* of a software layer is how much write traffic the layer produces in order to process a unit of incoming data.

This work is supported by research grant MOST 104-2221-E-009-011-MY3 from Ministry of Science and Technology, Taiwan, R.O.C.

LSM trees employ *compaction* of SSTables to propagate new KV pairs from higher levels down to lower levels. The compaction process is responsible for the first-stage write amplification. LSM trees store SSTables in files, which are allocated in terms of file-system blocks. However, flash pages are much larger than file system blocks (typically, 32 KB vs. 4 KB). Writing SSTable files may partially update a flash page, introducing the second-stage write amplification through page *read-modify-write*. The underlying SSDs employ a firmware layer, called FTL, to interact with the host as if they were ordinary hard drives. Because NAND flash is a kind of erase-before-write non-volatile memory, the FTL must timely perform *garbage collection*, which involves a series of data copying and flash erasure, to reclaim free space for writing. The FTL creates the third-stage write amplification through garbage collection. The final write traffic arriving at flash memory is the total size of the written KV pairs *multiplied* by all these three write amplifications.

For example, suppose that the write amplifications of the LSM tree layer, the block layer, and the FTL layer are 10, 1.5, and 3, respectively. Because of the compaction overhead, to handle one unit of application data, on average the LSM tree writes ten units of data to the block layer. Next, for each one of the ten units of data arriving at the block layer, on average the block layer writes 1.5 units of data to the FTL due to the read-modify-write overhead. For each one of the 1.5 units of data to the FTL, on average the FTL writes 3 units of data to flash memory due to the garbage collection overhead. Summing up, on average the software stack writes $10\times1.5\times3=45$ units of data to flash to handle one unit of data from applications.

In this study, we propose the design of KVSSD, which is based on a close integration of LSM trees and the FTL. A KVSSD employs a flash-native implementation of LSM trees, and it interacts with the host through KV operations rather than block I/O commands. The primary design goal is to eliminate or to reduce the write amplifications from different software layers: Firstly, we upgrade the existing logical-to-physical (L2P) mapping of the FTL to key-to-physical (K2P) mapping for efficient indexing. Because the K2P mapping decouples the logical order from the physical order of KV pairs, it is then a nature extension to implement copy-less SSTable compaction through remapping of KV pairs. Secondly, by allocating SSTables in terms of flash pages, the overhead of read-modify-

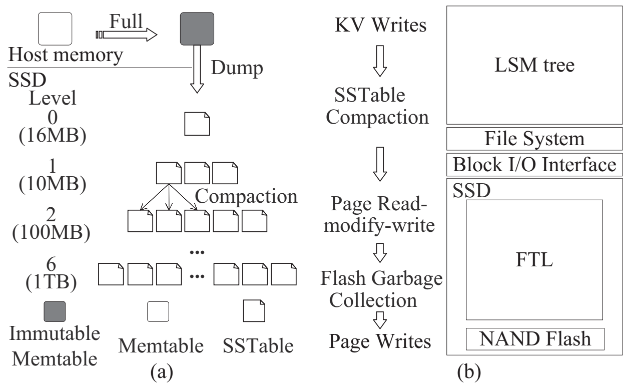

Fig. 1. (a) An LSM tree and SSTable compaction. (b) The software stack of the conventional LSM-on-SSD design. The LSM tree, the block layer, and the FTL induce cascading write amplifications.

write for flash pages can be completely eliminated. Thirdly, LSM tree levels have different capacity limits, and SSTables at higher levels undergo compaction more frequently than those at lower levels do. Exploiting this property, we propose to write pages of SSTables from different levels to different flash blocks. This write strategy actually implements hot-cold separation, which is an essential technique to reduce the write amplification of garbage collection [11].

Our experimental results show that, compared to conventional LSM-on-SSD methods, our LSM-SSD integrated approach improved the write amplifications caused by SSTable compaction, page read-modify-write, and flash garbage collection by 38%, 31%, and 72%, respectively, reducing the overall write amplification by 88% and improving the KV operation throughput by 347%.

II. Background

A. KV-Store Software Stack

LSM Trees. Figure 1(a) shows an LSM tree. The LSM tree collects application-generated KV pairs in main-memory using Memtables, and it flushes Immutable (full) Memtables to the storage as SSTables at proper timing. The LSM tree organizes SSTables in a hierarchical structure, and it always adds new SSTables to the top level (level 0). The key range of an SSTable is the interval between the largest and the smallest key of the SSTable. The capacity of the current level is ten times as small as that of the next lower level. Now, when a level is full, the LSM tree selects an SSTable from this level as a victim and performs compaction on the victim and the SSTables in the next level that overlap with the victim in terms of key range. The compaction process is essentially merge sorting on the involved SSTables. It writes new disjoint SSTables with sorted KV pairs to the next level and then deletes the old SSTables. Invalidated or deleted KV pairs are discarded during compaction.

FTL. NAND flash is erase-before-write. It reads and writes in terms of 32 KB pages, and an erasure unit typically has 128 pages. To avoid block erasure on every in-place page update, the FTL updates page in an out-of-place manner and uses L2P mapping to redirect I/O requests to the newest versions of data. Page updates consume free space, and the FTL reclaims pages occupied by stale data through migration of valid data followed by block erasure. This reclaiming process is called garbage collection.

Cascading Write Amplifications. Figure 1(b) shows the software stack of the conventional LSM-on-SSD approach. Applications call the LSM tree through get(), put(), and delete(). The LSM tree runs on top of a file system and produces block I/O writes. The LSM tree amplifies the block write traffic through SSTable compaction. When block I/O requests arrive at the FTL, many of them partially update flash pages because file-system blocks are smaller than pages. To handle a partial update to a page, the FTL first reads the old page, merges the old page with new data, and then writes a new page. This read-modify-write step amplifies the page write traffic. Next, the FTL writes new pages to garbage-collected free space. The garbage collection procedure involves migration of valid data and erasure of blocks, and it amplifies the total number of page writes. The overall write amplification is the product of the three write amplifications.

B. Related Work

LSM trees are a write-optimized index structure thanks to their sequential-logging behavior. However, the out-of-place logging design necessitates compaction, which noticeably amplifies the write traffic. Several approaches have been introduced to reduce the write amplification due to compaction: Wu et al. [10] and Yao et al. [12] proposed to divide the victim SSTable into small pieces and append the pieces to the lower-level SSTables that overlap the victim in terms of key range. Pan et al. proposed to delay SSTable compaction by writing virtual SSTables, which are an extra indirection to SSTables, instead of writing new SSTables [13]. These techniques, however, consider neither write amplifications from other software layers nor the existing flash management facilities like L2P mapping.

Recent KV store designs started to exploit SSD internal architectures. Marmol et al. proposed NVMKV, which upgrades the one-to-one L2P mapping to a sparse L2P mapping [6]. KV pairs are first hashed to a huge logical space, which is then sparsely mapped to flash memory. However, hash tables inherently generate small, random writes to flash memory, causing high read-modify-write and garbage collection overheads. Wang et al. [14] proposed LOCS, which aims at exploiting the chip parallelism of the Open-Channel SSD platform to parallelize the compaction process. However, LOCS does not attempt to optimize the write amplification of SSTable compaction. Shen et al. proposed DIDA, which is also based on the Open-Channel SSD platform. DIDA stores a hash table in the host memory for K2P mapping and uses slab allocation to reduce internal fragmentation of flash blocks [4]. However, the L2P hash table can be extremely large and does not fit in the embedded RAM of SSDs. Chen et al. aims at eliminating internal fragmentation of flash blocks [5]. They proposed to use multiple slab caches of different allocation sizes (all powers of 2) and partition a KV pair among these slab caches. However, this approach significantly increases the read overhead because a KV pair is scattered over multiple

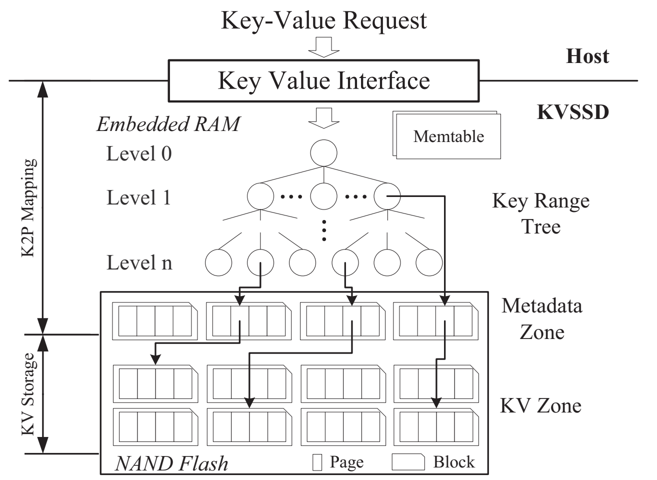

Fig. 2. A design overview of a KVSSD based on the proposed nSLM tree.

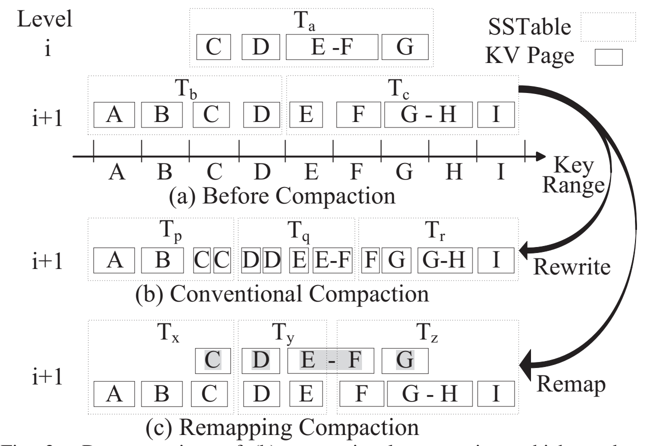

Fig. 3. Demonstrations of (b) conventional compaction, which produces new SSTables T_p, T_q, and T_r by writing 12 KV pages and (c) remapping compaction, which produces T_x, T_y, and T_z by remapping existing KV pages.

pages. These prior approaches exploit architectural supports of SSDs, but they do not consider the synergy between the management of KV data structure and that of flash memory.

III. A Close Integration of LSM Trees and FTL

A. Overview

Figure 2 is an overview of the proposed KVSSD. The KVSSD interacts with the host through KV operations. It employs a flash-native implementation of LSM trees, called the NAND-flash-LSM (nLSM) tree. The nLSM tree replaces the FTL L2P mapping table with a key-range tree, and each tree node represents the key range of an SSTable. A tree node contains a pointer to a flash page that stores the metadata of an SSTable, called a metadata page. The nLSM tree partitions the entire flash into a KV zone and a metadata zone. A page in the KV zone (a KV page) stores sorted KV pairs, and a page in the metadata zone (a metadata page) contains only pointers to KV pages and their key ranges. The nLSM tree optimizes write amplifications using different techniques: 1) It allocates flash pages to SSTables to eliminate read-modify-write operations, 2) it remaps metadata pages to KV pages for copy-free SSTable compaction, and 3) it separates KV pages from different tree levels to improve the garbage collection efficiency. These techniques will be explained in the following sections.

B. K2P Mapping and SSTable Allocation

K2P mapping finds the exact physical flash address of a target KV pair. The resolution of K2P mapping is much higher than that of L2P mapping, so it is nearly impossible to store the entire K2P mapping information in the SSD embedded RAM. We propose to store only the SSTable-level K2P mapping information in the embedded RAM using a key-range tree. The key-range tree represents the structure of an LSM tree, and a key-range node contains nothing but the key range of an SSTable and the physical page address (PPA) of the metadata page of the SSTable. An SSTable stores sorted KV pairs in KV pages, whose key ranges are all disjoint. The metadata page of an SSTable contains PPAs and key ranges of the KV pages of the SSTable. Now, to locate a KV pair, the nLSM tree finds a metadata page using the key-range tree, locates a KV page using the key-range information in the metadata page, and then retrieves the target KV pair from the KV page.

nLSM trees exclusively allocate a flash block to an SSTable. In our current design, the flash block size is 4 MB, which is as large as the SSTable size. By this design, nLSM trees eliminate the overhead of page read-modify-write because SSTables are always physically contiguous and aligned to page boundaries. nLSM trees inherit the compaction procedure from LSM trees, and thus during compaction, they always write in terms of SSTables. When an nLSM tree compacts a set of old SSTables, it performs multi-way merge sorting on these old SSTables and writes new SSTables of sorted KV pairs. After compaction, the old SSTables are discarded and their flash blocks contain all invalid data (garbage). Later on, flash garbage collection simply erases these blocks without any data copying.

C. Remapping Compaction

An LSM tree manages tree level sizes through compaction. Figure 3(a) shows that SSTable T_a at level i is a victim of compaction. The two SSTables T_b and T_c at level $i+1$ overlap T_a in terms of key range. Figure 3(b) shows that the LSM tree performs three-way merge sorting on T_a, T_b, and T_c, writes three new SSTables T_p, T_q, and T_r to level $i+1$, and deletes the old SSTables T_a, T_b, and T_c. There is no key-range overlap among the new SSTables T_p, T_q, and T_r. This conventional compaction process rewrites 12 KV pages plus 3 metadata pages of the three new SSTables.

With our K2P mapping, the logical order of KV pages is decoupled from their physical order in flash. It is therefore a nature extension to implement SSTable compaction through K2P remapping. We propose *remapping compaction*, and Figure 3(c) shows how it works: Remapping compaction writes three metadata pages to create three new SSTables T_x, T_y, and T_z and then deletes the metadata pages of T_a, T_b, and T_c. The new metadata pages contain new pointers to the *existing* KV pages. By this design, remapping compaction writes only 3 pages instead of 15 pages.

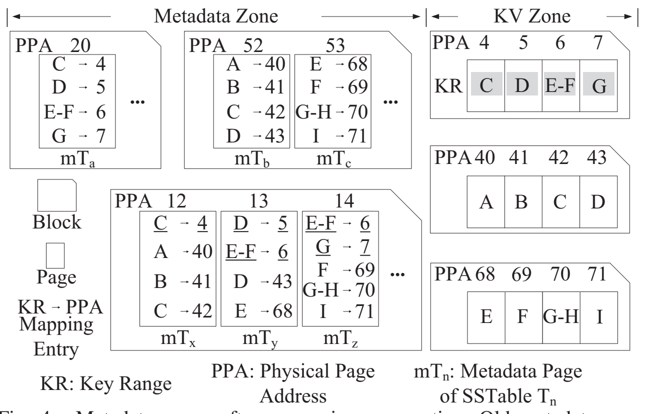

Fig. 4. Metadata pages after remapping compaction. Old metadata pages mT_a, mT_b, and mT_c will be deleted. Underlined mapping entries in new metadata pages mT_x, mT_y, and mT_z point to overlap pages (shaded ones).

Because remapping compaction does not rewrite KV pairs, it may create limited key-range overlaps among pages within an SSTable. For example, two among the pages in T_x are previously from different levels, and they both cover key range C. If two pages in an SSTable overlap each other in terms of key range, then the page previously from an upper level is an *overlap page*. In Figure 3(c), overlap pages are marked in gray. It is also possible that an overlap page is shared between two SSTables if the key boundary between the two SSTables is within the shared overlap page. For example, T_y and T_z share the same overlap page that covers key ranges E and F. A shared overlap page is included in the capacity of the leftmost involving SSTable, T_y in this case. Figure 4 shows the metadata pages and KV pages after the remapping compaction. The new metadata page mT_x contains four entries, one for the overlap page at PPA 4, which was previously associated with T_a, and the other three for disjoint KV pages at PPAs 40, 41, and 42, which were previously associated with T_b.

Let an *rLSM* tree be an nLSM tree enhanced by remapping compaction. An rLSM tree writes fewer pages on compaction than an nLSM tree does, but it may amplify page reads on query. This is because if the target key is within the key range of an overlap page, then the overlap page must be examined before the other pages in the same SSTable. To manage read amplification, an SSTable maintains a counter that indicates how many different levels its pages are from. For example, the counter of T_a is 1 and that of T_x is 2. Now, during compaction, if the counter of a new SSTable is less than or equal to a predefined threshold TH, then the new SSTable will be produced by remapping compaction. Otherwise, the new SSTable will be produced using conventional compaction, and the counter of the new SSTable will be reset to 1.

D. Hot-Cold Separation and Garbage Collection

Because conventional compaction writes in terms of SSTables, it produces all-invalid blocks for erasure. However, if remapping compaction is employed, the KV pages in the same block can be remapped to new SSTables at different levels. These new SSTables will later be involved in conventional compaction at different time points and leave some invalid pages in the block. As a result, the block will contain both valid and invalid pages, and garbage collection has to migrate valid pages from the block before erasing it.

Efficient garbage collection relies on separating data of different lifetimes among flash blocks [11]. This technique, also called hot-cold separation, is an essential technique to reduce the garbage collection write amplification. In LSM trees, because upper levels are smaller than lower levels, SSTables at upper levels are involved in compaction more frequently. In other words, pages associated with different levels have different lifetimes (or hotness). Based on this property, when garbage collection is migrating valid pages from a victim block, we propose to write KV pages mapped to the same level to the same flash block. By this design, pages of similar lifetimes are grouped in the same block for hot-cold separation.

In rLSM trees, garbage collection always erases the block of the fewest valid pages. Before erasing a block, garbage collection migrates all valid pages from it. To reflect the new PPAs of the migrated pages, garbage collection reversely traces all the SSTables previously associated with the migrated pages, rewrites the metadata pages of these SSTables, and updates the key-range nodes of these SSTables. For the reversal tracing, each block has a list of associated SSTables. We also add a reference count to every page to safely delete a shared overlap page created by remapping compaction. For example, T_y and T_z share an overlap page whose reference count is 2. When an SSTable is deleted, the reference counts of all its pages are decremented by one. A page becomes invalid when its reference count becomes zero.

E. Overhead Analysis

The embedded RAM space requirement of an rLSM tree includes the key-range tree, the per-block SSTable lists, and the per-page reference counters. The key-range tree size is related to the total number of SSTables and is a block-level overhead. The SSTable list length and the reference counter size depend on the remapping-compaction threshold TH. The larger the TH is, the longer an SSTable list is and the larger a reference count is. To limit the space requirement of the lists and the counters, in our current design, when an rLSM tree tries to create a new SSTable through remapping compaction, if any of the involved KV pages whose reference count is 7 (represented by 3 bits) or any of the involved block whose SSTable list length is 5, then the rLSM tree uses conventional compaction instead. By this design, the aforementioned data structures of a 16GB KVSSD require only about 2MB of embedded RAM.

IV. Experimental Results

A. Experimental Setup and Performance Metrics

For performance evaluation, we used Yahoo Cloud Serving Benchmark (YCSB) [15] to synthesize our KV-store workloads. The workload generation involved a loading phase, which inserted ten million unique KV pairs, and a write-intensive transaction phase, which involved insertion, update, and read (query) operations at rates of 25%, 65%, and 10%,

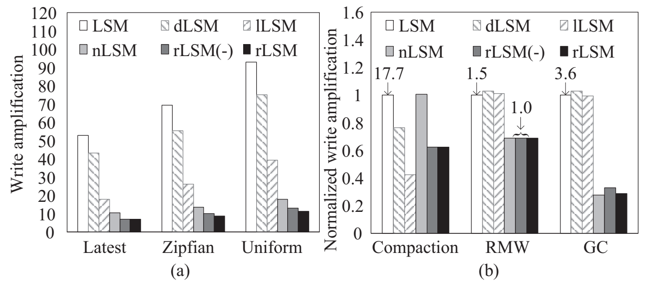

Fig. 5. (a) Overall write amplification. (b) Write amplifications contributed by SSTable compaction, page read-modify-write, and flash garbage collection under the Uniform key distribution. The numbers above arrows indicate the absolute values of write amplification.

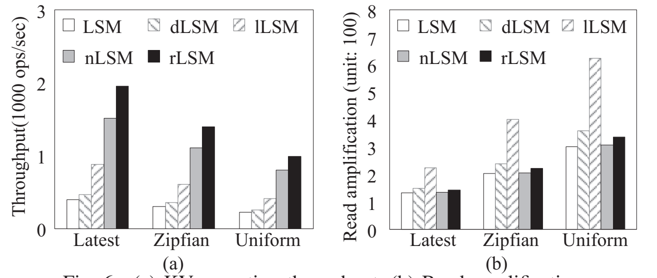

Fig. 6. (a) KV operation throughput. (b) Read amplification.

respectively. These parameters were selected to reflect write-intensive workloads. Different workloads were generated using the Latest, Zipfian, and Uniform key distributions. The key and value size were 24 and 1000 bytes, respectively.

Our experiments involved two major components: the LSM tree and FTL. We employed the LSM tree implementation of LevelDB [3] and the FTL implementation in the SSD Extension of DiskSim [16]. Our evaluation involved a family of LSM-on-SSD approaches: **LSM**, which stands for regular LSM trees. We replayed the KV workloads on LevelDB and collected block I/O traces using *blktrace*, and then replayed the traces on the SSD simulator to collect statistics of flash operations. **dLSM**, which is LSM enhanced by delayed compaction [13]. **lLSM**, which is LSM enhanced by lightweight compaction [12]. See Section II-B for the details of dLSM and lLSM. We also evaluated a family of our LSM-SSD integrated approaches: **nLSM**, which is equipped with the proposed K2P mapping and block-based SSTable allocation. We ported the core logic of LSM trees from LevelDB to the SSD simulator and ran the KV workloads on the KVSSD simulator to collect flash operation statistics. **rLSM(-)**, which enhances nLSM with remapping compaction. **rLSM**, which adds hot-cold separation to rLSM(-). rLSM is the fully-fledged version of our approach. In our experiments, the flash capacity was 15 GB, of which 5% was reserved as over-provision. The page and block size were 32 KB and 4 MB, respectively.

Our experiments employed three major performance metrics: 1) Write amplification, which was separately computed for LSM tree compaction, page read-modify-write, and flash garbage collection. 2) Throughput, which stands for the number of completed KV operations (including read and write) per second. 3) Read amplification, which is the average number of bytes read from flash to process one byte of KV read (query). Read amplification involved KV read operations only. The most efficient KV store design delivers the highest throughput while incurring the smallest read and write amplification.

B. Write Amplification

Figure 5(a) shows the overall write amplification. Generally, all the methods had a lower write amplification under the Latest key distribution (Latest for short) than they did under Uniform. This is because Latest exhibited better temporal localities of KV updates and therefore SSTable compaction removed many outdated KV pairs.

The LSM-SSD-integration family (nLSM, rLSM(-), rLSM) significantly outperformed the original LSM. The performance advantages are revealed in Figure 5(b), which separately shows the write amplifications contributed by SSTable compaction, page read-modify-write, and flash garbage collection under Uniform. The product of the three is the overall write amplification. For example, the overall one of LSM under Uniform is 93, which closely matches the product of the cascading write amplifications of LSM $17.7\times1.5\times3.6=95.6$. In Figure 5(b), nLSM and LSM performed nearly the same in terms of compaction write amplification, because they employed the same compaction design. nLSM exclusively allocated a flash block to an SSTable, so it produced all-invalid blocks by rewriting SSTables during compaction. Compared to LSM, nLSM eliminated the write amplification of read-modify-write (from 1.5 to 1) and that of garbage collection (from 3.6 to 1), significantly reducing the overall write amplification by 81%. rLSM(-) improved upon nLSM using remapping compaction, further reducing the compaction write amplification by 38%. This is because our remapping compaction writes a few metadata pages instead of rewriting SSTables. However, rLSM(-) slightly increased the pressure on garbage collection because it mixed KV pages that are mapped to different levels in the same block. rLSM resolved this issue by hot-cold separation. Compared to rLSM(-), rLSM reduced the write amplification due to garbage collection by 14%, and this improvement was reflected on the overall write amplifications of rLSM(-) and rLSM (13.1 vs 11.4).

dLSM delays conventional compaction by writing virtual SSTables, which contain extra indirection to existing SSTables. However, dLSM did not improve the compaction write amplification as much as rLSM did. This is because, when removing a virtual SSTable, dLSM must rewrite all the SSTables associated with the virtual SSTable. By contrast, every remapping-compacted SSTable can be individually involved in future compaction. lLSM achieved the best compaction write amplification. This is because lLSM compaction always appended KV pairs to existing SSTables instead of writing new SSTables. However, the price paid for this design is the poor space utilization. As reported in [12], *the overall space utilization of this compaction design is as low as 65%*, because SSTables reserve large space for appending new KV pairs. In our experiments, lLSM could not complete the tests unless we increased the SSD volume size by about 30%. LSM, dLSM, and lLSM all suffered from a high write amplification of read-modify-write, because they generated small requests to write

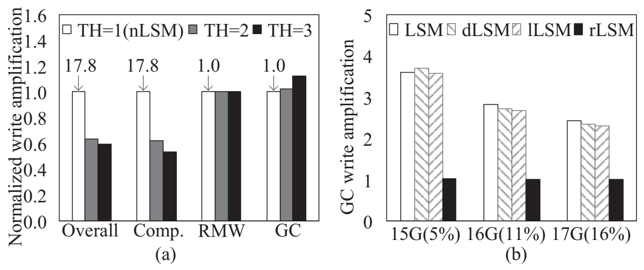

Fig. 7. (a) Normalized write amplifications under different remapping thresholds. The numbers above arrows indicate the absolute values of write amplification. (b) Write amplification of garbage collection under various flash over-provisions.

SSTable metadata and fragmented SSTable files through the file system layer. They also had a high write amplification of garbage collection because they had no access to data allocation in flash and therefore they mixed hot and cold data in blocks.

C. Throughput and Read Amplification

Figure 6(a) shows the throughput of all methods. Basically, each throughput is inversely proportional to its corresponding overall write amplification. In particular, the throughput of rLSM was up to 4.47 times as high as that of LSM. The improvements in throughput were slightly lower than those in overall write amplifications because the throughput includes KV read operations. Figure 6(b) reports the read amplifications of all methods. Read amplifications under Latest were lower than those under Uniform. This is again because Latest exhibited stronger temporal localities of KV operations, and popular KV pairs can be found at higher LSM tree levels. Notice that rLSM experienced an up to 11% increase in read amplification compared to LSM, and the extra read traffic was contributed by overlap pages in remapping-compacted SSTables. dLSM had an even higher increase in read amplification compared to LSM, reaching up to 19%. This is because, to search a KV pair in a virtual SSTable, dLSM had to read the metadata of all the associated real SSTables and the overlap segments in these SSTables. This problem became the worst in lLSM, because an SSTable may contain a large number of appended segments of KV pairs, and many of their key ranges overlap each other.

D. Remapping Threshold and Overprovision

We evaluated rLSM with various settings of the remapping-compaction threshold TH. Figure 7(a) shows that, as TH increased, the extra reduction in compaction write-amplification diminished. This is because the frequency of conventional compaction is inversely proportional to TH, and the change in the compaction frequency becomes small among large TH values. A large TH also damaged the efficiency of garbage collection because KV pages in a block may be remapped to new SSTables at many different levels, exaggerating the mixture of hot and cold data in blocks. The results suggest that 2 will be a good choice for TH.

We observed how flash over-provision size impacted on write amplification of garbage collection. Notice that changing the over-provision size will not affect the write amplifications of compaction and read-modify-write. Figure 7(b) shows that LSM, dLSM and lLSM demanded large flash over-provision to relieve the negative impact caused by mixing hot and cold data. By contrast, rLSM employed hot-cold separation, so its garbage collection was very efficient even under the smallest over-provision size.

V. CONCLUSIONS

The conventional LSM-on-SSD approach provides excellent modularity, but it suffers from cascading write amplifications contributed by SSTable compaction, page read-modify-write, and flash garbage collection. Many of the existing studies are focused on compaction overhead optimization, but they pay little attention to the write amplifications from other software layers. To manage the overall write amplification, we propose a close integration of LSM trees and the FTL. Specifically, we propose copy-free remapping compaction, flash-aware data allocation, and hot-cold separation to optimize the write amplifications of compaction, read-modify-write, and garbage collection, respectively. Results show that our integrated approach significantly outperformed existing work in terms of KV operation performance.

REFERENCES

[1] B. Debnath, S. Sengupta, and J. Li, "FlashStore: high throughput persistent key-value store," *Proc. VLDB Endow.*, vol. 3, no. 1-2, 2010.
[2] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: amazon's highly available key-value store," *ACM SIGOPS operating systems review*, vol. 41, no. 6, 2007.
[3] LevelDB., "https://github.com/google/leveldb," 2017.
[4] Z. Shen, F. Chen, Y. Jia, and Z. Shao, "DIDACache: A deep integration of device and application for flash based key-value caching," in *USENIX FAST*, 2017.
[5] Y.-T. Chen, M.-C. Yang, Y.-H. Chang, T.-Y. Chen, H.-W. Wei, and W.-K. Shih, "KVFTL: Optimization of storage space utilization for key-value-specific flash storage devices," in *IEEE ASP-DAC*, 2017.
[6] L. Marmol, S. Sundararaman, N. Talagala, and R. Rangaswami, "NVMKV: A scalable, lightweight, FTL-aware key-value store." in *USENIX ATC*, 2015.
[7] A. Papagiannis, G. Saloustros, P. González-Férez, and A. Bilas, "Tucana: Design and implementation of a fast and efficient scale-up key-value store." in *USENIX ATC*, 2016.
[8] P. ONeil, E. Cheng, D. Gawlick, and E. ONeil, "The log-structured merge-tree (LSM-tree)," *Acta Informatica*, vol. 33, no. 4, 1996.
[9] L. Lu, T. S. Pillai, H. Gopalakrishnan, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "WiscKey: Separating keys from values in SSD-conscious storage," *ACM TOS*, vol. 13, no. 1, 2017.
[10] X. Wu, Y. Xu, Z. Shao, and S. Jiang, "LSM-trie: an LSM-tree-based ultra-large key-value store for small data," in *USENIX ATC*, 2015.
[11] L.-P. Chang, Y.-S. Liu, and W.-H. Lin, "Stable greedy: Adaptive garbage collection for durable page-mapping multichannel SSDs," *ACM TECS*, vol. 15, no. 1, 2016.
[12] T. Yao, J. Wan, P. Huang, X. He, Q. Gui, F. Wu, and C. Xie, "A light-weight compaction tree to reduce I/O amplification toward efficient key-value stores," in *IEEE MSST*, 2017.
[13] F. Pan, Y. Yue, and J. Xiong, "dCompaction: Delayed compaction for the LSM-tree," *International Journal of Parallel Programming*, 2016.
[14] 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 *ACM EuroSys*, 2014.
[15] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with YCSB," in *ACM SoCC*, 2010.
[16] Microsoft Research. SSD extension for DiskSim simulation environment. [Online]. Available: http://research.microsoft.com/en-us/downloads/b41019e2-1d2b-44d8-b512-ba35ab814cd4/