Isolating Namespace and Performance in Key-Value SSDs for Multi-tenant Environments

Donghyun Min and Youngjae Kim
Dept. of Computer Science and Engineering, Sogang University, Seoul, Republic of Korea
{mdh38112,youkim}@sogang.ac.kr

ABSTRACT

Key-value SSDs (KVSSDs) implement the storage engine of a key-value store such as log-structured merge-tree (LSM-tree) inside the SSD. However, recent LSM-tree based KVSSDs cannot be used directly in a multi-tenant environment. LSM-tree-based KVSSDs are not designed with isolation in mind in terms of namespaces and performance, leading to incorrect data access between concurrent users and poor read performance. In this paper, we propose Iso-KVSSD, a LSM-tree based KVSSD for multi-tenancy by supporting namespace and performance isolation. The Iso-KVSSD performs access control based on the user's namespace and constructs per-namespace dedicated LSM-trees for users. We implement the Iso-KVSSD on Cosmos+ OpenSSD in a Linux environment and evaluate performance with Put() and Get() workloads by varying the number of tenants. Our extensive evaluation results showed that Iso-KVSSD has negligible write performance overhead and an average 2.9 times higher read throughput than a baseline that manages one global shared LSM tree between users.

1 INTRODUCTION

Key-value stores (KV-stores) such as RocksDB [7], LevelDB [6], and MongoDB [5] are NoSQL databases running on a host. On the other hand, key-value solid-state drives (KVSSDs) [3, 4, 10, 12, 14, 19] run the storage engine of a KV-store on the SSD. Recently, KVSSDs with a LSM-tree based indexing approach have emerged [10, 14, 19]. LSM-tree [16] is a data structure that performs out-of-place logging to disk sequentially instead of in-place write when storing KV data. Thus, LSM-tree-based KVSSDs provide optimized performance for write-intensive workloads.

HotStorage'21, July 27-28, 2021, Virtual
© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-8550-3/21/07…$15.00
https://doi.org/10.1145/3465332.3470883

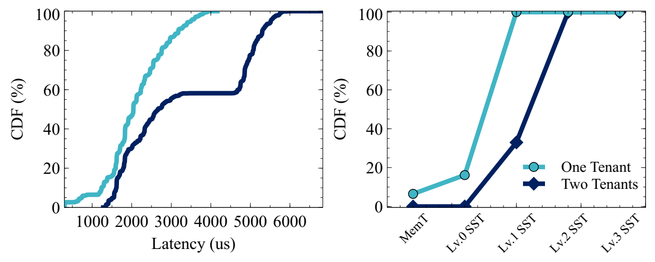
Multi-tenancy is an architecture that can host multiple database instances of tenants on a server. In a multi-tenant environment, concurrent users are provided with an abstraction of having their own dedicated server, requiring isolation in terms of security, privacy, and performance [8]. The aforementioned requirements for isolation can be met based on namespace isolation. Namespace is a granularity that separates the KV data of tenants stored in physical space into logical groups. Several existing KV-stores such as RocksDB and MongoDB have been designed to be multi-tenant [13, 18].

Several studies on LSM-tree based KVSSDs [10, 14, 19] even lack design and implementation for namespace isolation. Therefore, each user fails to be provided a strict view showing only the data corresponding to their own namespace. Accordingly, data from other users can be easily modified or read recklessly [11, 17]. Furthermore, current LSM-tree-based KVSSDs have difficulty in providing the promised read throughput that the storage device can provide for each tenant in a multi-tenant environment. This is because multiple KV data from tenants are still managed by a single global LSM-tree index structure.

Specifically, the global shared LSM-tree structure of the KVSSD has the following limitations. First, it is likely that most tenants' KV data will be indexed at the upper level of the LSM-tree. So, traversing the LSM tree is time consuming because the search algorithm starts at the lowest level of the index. Second, to check the existence of the requested key during LSM-tree search, it must read the Bloom filter (BF) data structures of the LSM-tree from the NAND flash memory several times.

In this paper, we propose Iso-KVSSD, a LSM-tree based KVSSD that supports isolation in terms of namespace and performance for multi-tenant environments. This paper makes the following specific contributions.

- For namespace isolation, Iso-KVSSD identifies the user's namespace information and performs namespace-based access control for each user. The namespace is stored in the NSID region of the NVMe command and then transferred to Iso-KVSSD.
- For performance isolation, Iso-KVSSD adopts a namespace dedicated LSM-tree design according to the user's namespace. Each user is given their own independent dedicated



(a) Response time (b) Level of accessed data

Figure 1: Comparative evaluation of the impact that tenant *x* receives from co-located neighbor tenant *y*.

LSM-tree. This is achieved by partitioning the global LSM-tree into per-user LSM-trees according to namespace. This approach does not require any additional total cost of ownership while delivering promising performance to tenants that the storage device can provide with little performance loss.

- Iso-KVSSD was implemented on the Cosmos+ OpenSSD Platform [1] in a Linux environment. Iso-KVSSD was evaluated and compared with baseline which adopts a global shared LSM-tree with workloads using Put() and Get() while increasing the number of tenants. Our extensive experiments showed that Iso-KVSSD improves read throughput by up to 2.9 times while write performance is barely different from the baseline.

2 BACKGROUND AND MOTIVATION

2.1 Log-Structured Merge-Tree

LSM-tree [16] is a hierarchical structure that consists of an internal DRAM component (MemTable) and a NAND flash memory component (SSTable) in KVSSDs. MemTable temporarily stores KV data transferred from the user. MemTable is mutable and is typically implemented as a skiplist. A SSTable is an immutable index file created when KV data is flushed and stored from MemTable to NAND flash memory. This process is called SSTable flushing. SSTable is structured into multiple levels (L_0, L_1,..., L_n).

Each level contains several key-sorted SSTables. In the LSM-trees following the key-value separation design proposed in WiscKey [15], each SSTable is composed of a meta (key, value log offset) region and Bloom filter (BF) region. The meta region consists of a key and an offset of associated value stored in the value log. BF is a space-efficient probabilistic data structure and is used to test whether an element is a member of a set. By leveraging BF, the existence of the target key can be verified without checking all the keys of the SSTable. The L_i SSTable acts as a buffer for L_{i+1}, which is larger in size compared to the L_i. LSM-tree triggers a compaction process when the KV data in L_i SSTable reaches a certain threshold in size. The compaction means the process of choosing at least two L_i victim SSTables, merging into a L_{i+1} SSTable, and sorting entries based on a key. After compaction, the older version of L_i SSTable entry is removed.

2.2 Motivation

Several recent LSM-based KVSSDs [10, 14, 19] lack design for namespace isolation. This causes the problem of KV data being incorrectly modified or read if tenants have the same key [11, 17]. Also, the design of such an LSM-based KVSSD is not sufficient for performance isolation. For example, in a traditional LSM-based KVSSD, multiple tenants share the LSM tree, so the read performance of each tenant cannot be guaranteed.

In order to quantitatively analyze the degradation of read performance of each tenant, we conducted experiments for the following scenarios: (i) When only tenant x's 1 M data occupies a LSM-tree, and (ii) When a LSM-tree is shared by tenant x's and tenant y's own 1 M data at the same time. The detailed experimental setup is the same as the settings described in Section 4.

Figure 1 shows the read performance of tenant x and the level at which KV data search occurs in the shared LSM-tree for both scenarios. Figure 1 (a) shows that average latency increased by 1.58x in scenario 2 compared to scenario 1. This is because all tenants' KV data are indexed in a global shared LSM-tree. In particular, there are two causes of performance interference between tenants. First, many KV data of each tenant are indexed at the higher level of the LSM tree. Figure 1 (b) shows that in scenario 2, the indexes of KV data of tenant x were pushed to the higher level in the LSM-tree compared to scenario 1. About 67 % of indexes of KV data of tenant x is searched in L_2 SSTable. Second, the task of loading BFs into DRAM multiple times from NAND flash memory during the search process incurs significant overhead. It was confirmed that the number of BF readings increased by about 77 % in Scenario 2 compared to Scenario 1.

With the aforementioned research motivations, this paper proposes Iso-KVSSD which aims to isolate namespace and performance in a multi-tenant environment while providing high read throughput and better response time. There may be security and privacy issues for each user in relation to namespace separation. The security and privacy issue are out of the scope of this study. In addition, Iso-KVSSD can implement an access authentication system for each user or a secure partition mechanism inside the SSD to support strong security such as DiskShield [2] and Inuksuk [20].

3 DESIGN AND IMPLEMENTATION

3.1 Problem Formulation

When KV data from each tenant are indexed to a single global LSM-tree, the time to access KV data can increase.

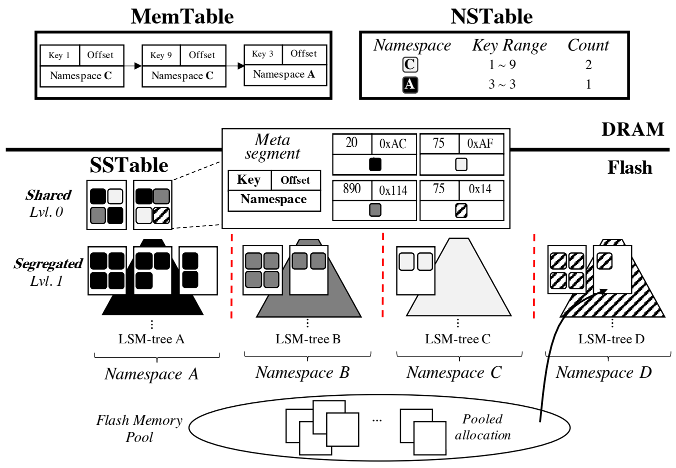


Figure 2: An architecture for Iso-KVSSD.

The first cause is that many KV data stored by each tenant may be pushed by data from other tenants and indexed at a higher level in the LSM tree. The second cause is that it requires loading the BF from NAND flash memory into memory multiple times during the LSM tree search process. The average latency for KV access (*ALKA*) can be estimated as a cost model for accessing KV data indexed in the LSM-tree. Table 1 shows a summary of the notations and their descriptions in the cost model. The *ALKA* is estimated by using the following Equation (1):

$$\begin{aligned} ALKA_i &= HR_i \cdot HT_i + (1 - HR_i)(MP_i + ALKA_{i+1}) \\ ALKA &= ALKA_0 + MemTbl \end{aligned} \tag{1}$$

In Equation 1, $ALKA_n = HR_n \cdot HT_n$ in the last tree level n, if `GET(k)` is not requested on the KVSSD where key k does not exist. This recurrence relation is modeled based on an insight of the average memory access time model [9] in the multilevel cache/memory hierarchical architecture. This is because LSM-tree is also a hierarchical structure of multilevel SSTables, and the KV data search algorithm begins at the lowest level SSTable in order. The first cause aforementioned corresponds to how much recursion is performed in this recurrence relation. Thus, it implies how much MP_n is overlapped. The second cause corresponds to the MP_n value itself in the equation.

Table 1: Notations used and their descriptions.

Notion	Description
n	Total possible level of LSM-tree
i	Index level of LSM-tree ($0 \le i \le n$)
HR_n	KV hit ratio in the level n SSTable
HT_n	KV hit time (combined latency of reading BF, meta, and value of level n SSTable from flash memory until key matches)
MP_n	KV miss penalty (combined latency of reading BF, meta of level n SSTables from flash memory)
$MemTbl$	MemTable access time

3.2 Per-namespace dedicated LSM Tree

Figure 2 shows the overall architecture of the Iso-KVSSD. First, for the index management of namespace isolation, namespace information is stored in the indexing structure along with the KV data. Namespace Table (NSTable) is a new memory component that stores the namespace information of the KV data in MemTable. As shown in Figure 2, NSTable keeps track of min, max keys per namespace and the total number of KV data per namespace. When KV data in the MemTable is made to SSTable after a flushing operation, the namespace and NSTable information per namespace are also stored in the SSTable footer. Accordingly, when reading KV data, access is allowed only if the matching of both the key and the namespace is satisfied.

Iso-KVSSD employs the per-namespace dedicated LSM-tree. In the per-namespace LSM-tree, MemTable and the lowest level SSTable (e.g., L_0) do not store KV data separately according to the namespace. They are mixed and stored in MemTable and L_0 SSTable. On the other hand, in the other levels (e.g., L_1,..., L_n) SSTable, KV data is segregated by its namespace and indexed to a different LSM-tree. This approach was designed for two reasons. First, the performance degradation caused by a mixture of data from any tenants in MemTable and L_0 SSTable is negligible. According to our experiment, the KV data are rarely searched in MemTable and L_0 SSTable (less than 3.2 %) compared to other level SSTables. The more the degree of multi-tenancy increases, the less KV data searched in MemTable and L_0 SSTable. Second, the MemTable provisioning per namespace approach requires additional DRAM space. If the number of concurrent tenants increases, more DRAM capacity is required in proportion to the degree of multi-tenancy.

The main expected effect of the namespace-dedicated LSM-tree is that *ALKA* is reduced. The first reason is that KV data from different namespaces are indexed to different LSM-trees, preventing them from being indexed to upper levels of LSM-trees. Thus, this mitigates the computation of MP_n by overlapping several times in the *ALKA* equation. In other words, the value of n becomes smaller. The second reason is that the number of BF readings from NAND flash memory decreases during data retrieval. This corresponds to a decrease in the MP_n value itself in the *ALKA* equation.

3.3 Namespace Isolation Mechanism

Figure 3 illustrates a namespace isolation mechanism that segregates KV data in a global LSM-tree into per-namespace dedicated LSM-trees based on a user's namespace. Namespace isolation is performed during the background compression process as in traditional LSM trees. The compaction process reads the victim SSTable from NAND flash memory and merge-sorts victim SSTables into one new SSTable. This

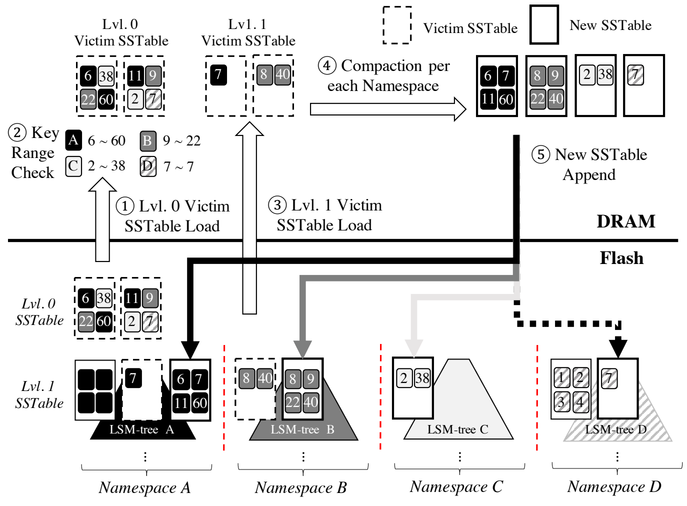


Figure 3: Per-namespace isolation process.

isolation strategy does not require an additional SSTable read operation because it performs isolation for the victim SSTable already read during compaction. Here, the namespace of each entry in the victim SSTable is called the victim namespace.

When compaction for L_0 SSTable starts, Iso-KVSSD first chooses the victim SSTables and reads them into DRAM memory (①). Since compaction is triggered at L_0, per-namespace isolation is enabled to classify KV data according to namespace. If compaction starts at a level other than L_0, the isolation task is disabled and performs the same as the traditional compaction flow. If isolation is enabled, the key range for each victim namespace in the L_0 victim SSTable is checked (②). This is done to verify whether the key range of the victim namespace is overlapped with the key range of the L_1 SSTable of the victim namespace LSM-tree. Then, L_1 SSTable with an overlapping key range is read into DRAM memory (③). Next, for each entry in the L_0 victim SSTable,compaction is performed with the corresponding namespace's L_1 victim SSTable (④). This process creates a new SSTable for each victim namespace. Finally, the indexing entry of multiple namespaces that existed in L_0 victim SSTable is appended by indexing to L_1 SSTable of the per-namespace LSM-tree (⑤).

Table 2: Key Value API for namespace spport.

API	Explanation
`ns = CREATE()`	Create and initialize the new LSM-tree and return associated namespace (*ns*) to user.
`DESTROY(ns)`	Clear contents of the LSM-tree corresponding to namespace (*ns*) and destroy the KV database.
`PUT(k, v, ns)`	Store the value (*v*) associated with the key (*k*) to a LSM-tree corresponding to the namespace (*ns*).
`GET(k, ns)`	Load the value (*v*) associated with the key (*k*) from the LSM-tree corresponding to the namespace (*ns*).

In the example shown in Figure 3, two L_0 victim SSTables are read during compaction. In this case, the L_1 SSTables of namespace *A* and *B* are read for isolation. For namespace *A*, the key range (7-7) of the second L_1 SSTable belongs to the key range (6 - 60) of the L_0 SSTable. For namespace *B*, the key range (9 - 22) of the L_0 SSTable belongs to the key range (8 - 40) of the first SSTable of L_1. A new L_1 SSTable is created by performing merge-sort for each namespace and is stored individually in the per-namespace LSM-tree.

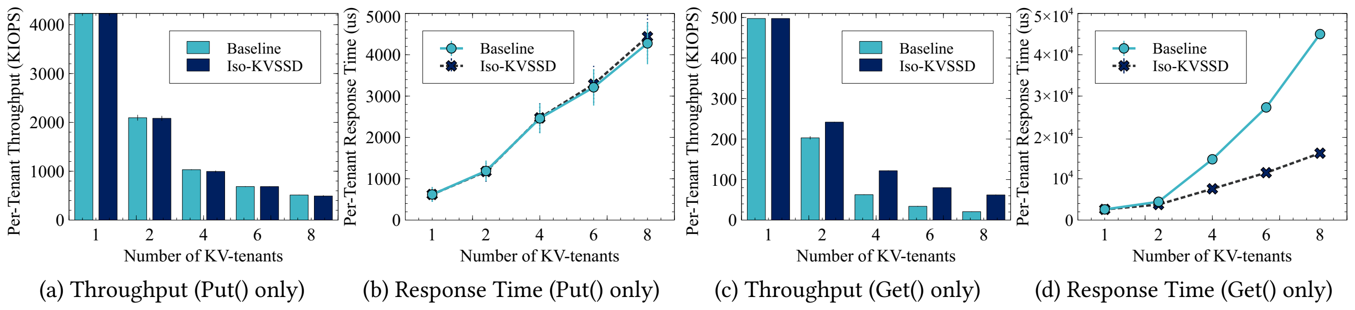
3.4 Key-Value API Library

Each user sends KV requests to Iso-KVSSD via the key-value API library in a host. The key-value API library for namespace support is described in Table 2. The key-value API uses a system call (e.g., ioctl) to pass KV data passed from the user to the Key-Value SSD kernel driver. Then, the kernel driver stores the namespace in the NSID region and key in the LBA region of the NVMe command. The memory address where the value is stored is recorded in the page list region of the NVMe command. Then, the NVMe command is sent to Iso-KVSSD.

4 EVALUATION

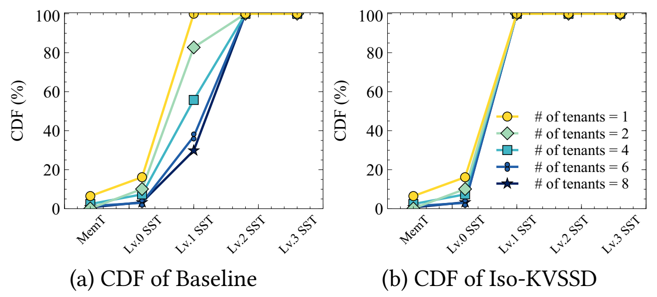
Experimental Setup: We prototyped Iso-KVSSD on the basis of iLSM [14] following the LSM-tree design of Wisckey [15]. The prototype environment was a FPGA-based Cosmos+ OpenSSD [1] equipped with 1GB DDR3 DRAM and ARM Cortex-A9 processors. Cosmos+ OpenSSD and host were communicated through the NVMe protocol. The default key and value sizes were set to 8 B and 1 KB, respectively, which represent averages of common KV workloads. During experiments, the number of concurrent users issuing KV requests, or degree of multi-tenancy, was increased from one to eight, and key was randomly specified. Each user issued the same number of 1 M Put() or Get() KV requests for Put() only and Get() only workloads. The baseline had namespace isolation enabled, but no performance isolation, which means that a LSM-tree was shared among users in the baseline. On the other hand, Iso-KVSSD had both namespace and performance isolation enabled.

Performance Comparison: We measured both per-tenant throughput and response time of Iso-KVSSD and baseline for Put() only and Get() only workloads. Figure 4(a) shows the negligible throughput difference between Iso-KVSSD and baseline for a Put() only workload. The throughput overhead of Iso-KVSSD was within 1 % compared to baseline regardless of the number of concurrent tenants. This slight overhead with Iso-KVSSD is due to the namespace isolation. Specifically, the isolation overhead is attributed to reading L_1 SSTable of the LSM-tree corresponding victim namespace.



(a) Throughput (Put() only) (b) Response Time (Put() only) (c) Throughput (Get() only) (d) Response Time (Get() only)

Figure 4: Throughput and average response time of Baseline and Iso-KVSSD with KV-tenants (1–8).



(a) CDF of Baseline (b) CDF of Iso-KVSSD

Figure 5: Level distribution of where KV data is indexed in the LSM-tree.

Also, Figure 4(b) shows little response time difference between the baseline and Iso-KVSSD. In particular, Iso-KVSSD shows only 4 % additional latency compared to baseline, when the number of tenant is eight.

In Figure 4(c), the Get() throughput difference between Iso-KVSSD and baseline is prominent. When the number of concurrent tenant is two, four, six, or eight, Iso-KVSSD shows 1.1, 1.9, 2.3, and 2.9 × higher Get() throughput than baseline, respectively. This is because Iso-KVSSD adopts the per-namespace LSM-tree. Therefore, Iso-KVSSD reduces the depth of LSM-tree as well as reduces the number of BF reads required during the KV data search process. Figure 4(d) shows a comparison of response times. The difference in average response time between baseline and Iso-KVSSD becomes evident as the number of tenants increases. Specifically, if the number of tenants is eight, the Iso-KVSSD has a 2.78 × lower average response time than baseline. We also experimented with the mixed workloads of Put() and Get() requests, but observed that their results were hardly different from those of Get() only workloads.

Impact of Per-namespace LSM-tree. Figure 5 represents a CDF on which the level of the LSM-tree indexing information is searched during Get(). Figure 5 (a) is a CDF of the baseline. When the number of tenants is one, KV data search is completed only with the indexing information of MemTable, L_0, and L_1 SSTable. However, as the number of tenants increases, the number of searches from the lower level index of the LSM-tree decreases and Get() is processed by searching the higher level index. This is due to the fact that all index information of tenants is managed by a global shared LSM-tree, thus forming a deeper level of LSM-tree. Specifically, as the number of tenants increases from one to eight, the percentage at which KV data is searched in L_1 SSTable is reduced from 83.8 % to 26.55 %, and the percentage at which KV data is searched in L_2 SSTable is increased from 0 % to 70.1 %. The percentage of data searched in MemTable and L_0 SSTable is only 3.2 %. On the other hand, Figure 5 (b) is a CDF of Iso-KVSSD. Since KV data from other namespaces are indexed to individual LSM-trees, KV data search is completed with only indexing information of MemTable, L_0, and L_1 SSTable regardless of the number of tenants. These results are evidence that a per-namespace LSM-tree can lower the number of recursions in *ALAK* Equation 1.

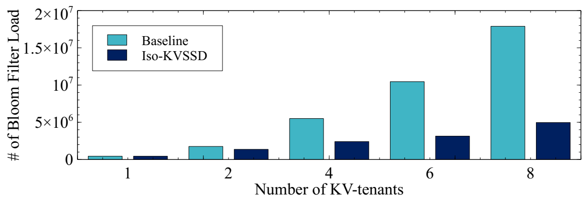


Figure 6: Bloom Filter load with KV-tenants (1–8).

Figure 6 represents how many BF loads are performed during Get(). Iso-KVSSD can reduce the number of BF loads due to the per-namespace LSM-tree. In particular, when the number of tenants is eight, Iso-KVSSD results in 3.6 × fewer BF loads than the baseline. These results are evidence that MP_n in *ALKA* Equation 1 can be reduced. This BF overhead can be further minimized by caching if there is enough DRAM inside the SSD.

5 CONCLUSION

We proposed Iso-KVSSD, which controls access to data based on a user's namespace. Iso-KVSSD implements per-namespace LSM-tree design and a namespace isolation mechanism. We prototyped Iso-KVSSD on Cosmos+ OpenSSD in a Linux environment and compared Iso-KVSSD with a baseline that uses a global LSM-tree. Extensive evaluation showed that read and write throughput of Iso-KVSSD was improved by up to 190% and decreased by less than 1% from baseline, respectively.

ACKNOWLEDGMENTS

We thank the reviewers and our shepherd, Janki Bhimani, for their constructive comments that have significantly improved the paper. This work was supported in part by a research grant from SK Hynix and by an Institute of Information communications Technology Planning Evaluation (IITP) grant funded by the Korea government (MSIT) (No. 2020-0-00104). Y. Kim is the corresponding author.

REFERENCES

[1] 2017. Cosmos+ OpenSSD Platform. http://www.openssd.io/.

[2] Jinwoo Ahn, Junghee Lee, Yungwoo Ko, Donghyun Min, Jiyun Park, Sungyong Park, and Youngjae Kim. 2020. DISKSHIELD: A Data Tamper-Resistant Storage for Intel SGX. In *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security (ASIACCS)*. ACM, 799–812.

[3] Janki Bhimani, Jingpei Yang, Ningfang Mi, Changho Choi, and Manoj Saha. 2021. Fine-grained Control of Concurrency within KV-SSDs. In *Proceeding of the 14th ACM International System and Storage Conference (Systor)*. ACM, 1–12.

[4] SAMSUNG ELECTRONICS. 2018. Samsung Smart SSD. https://samsungatfirst.com/smartssd-ocp/.

[5] Storage Engines. 2020. MongoDB Manual. https://docs.mongodb.com/manual/.

[6] Facebook. 2017. LevelDB. https://github.com/google/leveldb.

[7] Google. 2012. RocksDB: A Persistent Key-Value Store for Fast Storage Environment. https://rocksdb.org.

[8] Ajay Gulati, Arif Merchant, and Peter J Varman. 2007. pClock: An Arrival Curve based Approach for QoS Guarantees in Shared Storage Systems. *ACM SIGMETRICS Performance Evaluation Review* 35, 1 (2007), 13–24.

[9] John L Hennessy and David A Patterson. 2011. *Computer Architecture: A Quantitative Approach.* Elsevier.

[10] Junsu Im, Jinwook Bae, Chanwoo Chung, Arvind, and Sungjin Lee. 2020. PinK: High-speed In-storage Key-value Store with Bounded Tails. In *Proceeding of the USENIX Annual Technical Conference (ATC)*. USENIX, 173–187.

[11] Shvetank Jain, Fareha Shafique, Vladan Djeric, and Ashvin Goel. 2008. Application-level isolation and recovery with solitude. In *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*. 95–107.

[12] Yanqin Jin, Hung-Wei Tseng, Yannis Papakonstantinou, and Steven Swanson. 2017. KAML: A Flexible, High-Performance Key-Value SSD. In *Proceeding of the IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 373–384.

[13] Willis Lang, Srinath Shankar, Jignesh M Patel, and Ajay Kalhan. 2013. Towards multi-tenant performance SLOs. *IEEE Transactions on Knowledge and Data Engineering* 26, 6 (2013), 1447–1463.

[14] Chang-Gyu Lee, Hyeongu Kang, Donggyu Park, Sungyong Park, Youngjae Kim, Jungki Noh, Woosuk Chung, and Kyoung Park. 2019. iLSM-SSD: An Intelligent LSM-Tree Based Key-Value SSD for Data Analytics. In *Proceeding of the 27th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. IEEE, 384–395.

[15] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Hariharan Gopalakrishnan, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. 2016. Wisckey: Separating Keys from Values in SSD-conscious Storage. In *Proceedings of the File and Storage Technologies (FAST)*. USENIX, 133–148.

[16] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. 1996. The Log-Structured Merge-Tree (LSM-tree). *Acta Informatica* 33, 4 (1996), 351–385.

[17] Sean Rhea, Brighten Godfrey, Brad Karp, John Kubiatowicz, Sylvia Ratnasamy, Scott Shenker, Ion Stoica, and Harlan Yu. 2005. OpenDHT: A Public DHT Service and Its Sses. In *Proceedings of the 2005 conference on Applications, technologies, architectures, and protocols for computer communications*. 73–84.

[18] David Shue, Michael J Freedman, and Anees Shaikh. 2012. Performance Isolation and Fairness for Multi-tenant Cloud Storage. In *Proceeding of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX, 349–362.

[19] Sung-Ming Wu, Kai-Hsiang Lin, and Li-Pin Chang. 2018. KVSSD: Close Integration of LSM Trees and Flash Translation Layer for Write-efficient KV Store. In *Proceeding of the Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 563–568.

[20] Lianying Zhao and Mohammad Mannan. 2019. TEE-aided Write Protection Against Privileged Data Tampering. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*. USENIX.