LeaFTL: A Learning-based Flash Translation Layer for Solid-State Drives

Jinghan Sun
UIUC
js39@illinois.edu

Shaobo Li
UIUC
shaobol2@illinois.edu

Yunxin Sun*
ETH Zurich
yunsun@student.ethz.ch

Chao Sun
Western Digital Research
chao.sun@wdc.com

Dejan Vucinic
Western Digital Research
dejan.vucinic@wdc.com

Jian Huang
UIUC
jianh@illinois.edu

ABSTRACT

In modern solid-state drives (SSDs), the indexing of flash pages is a critical component in their storage controllers. It not only affects the data access performance, but also determines the efficiency of the precious in-device DRAM resource. A variety of address mapping schemes and optimizations have been proposed. However, most of them were developed with human-driven heuristics.

In this paper, we present a learning-based flash translation layer (FTL), named LeaFTL, which learns the address mapping to tolerate dynamic data access patterns via linear regression at runtime. By grouping a large set of mapping entries into a learned segment, it significantly reduces the memory footprint of the address mapping table, which further benefits the data caching in SSD controllers. LeaFTL also employs various optimization techniques, including out-of-band metadata verification to tolerate mispredictions, optimized flash allocation, and dynamic compaction of learned index segments. We implement LeaFTL with both a validated SSD simulator and a real open-channel SSD board. Our evaluation with various storage workloads demonstrates that LeaFTL saves the memory consumption of the mapping table by 2.9× and improves the storage performance by 1.4× on average, in comparison with state-of-the-art FTL schemes.

CCS CONCEPTS

• **Hardware → External storage**; • **Computer systems organization → Architectures**; • **Computing methodologies → Learning linear models**.

KEYWORDS

Learning-Based Storage, Flash Translation Layer, Solid-State Drive

*Work done when visiting the Systems Platform Research Group at UIUC as a research intern.

ASPLOS '23, March 25–29, 2023, Vancouver, Canada

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/XXXXXXX.XXXXXXX

ACM Reference Format:
Jinghan Sun, Shaobo Li, Yunxin Sun, Chao Sun, Dejan Vucinic, and Jian Huang. 2023. LeaFTL: A Learning-based Flash Translation Layer for Solid-State Drives. In *Proceedings of Architectural Support for Programming Languages and Operating Systems (ASPLOS '23), March 25–29, 2023, Vancouver, Canada.* ACM, New York, NY, USA, 15 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Flash-based SSDs have become an indispensable part in modern storage systems, as they outperform conventional hard-disk drives (HDDs) by orders of magnitude, and their cost is close to that of HDDs [22, 30, 51, 62]. The SSD capacity continues to boost by increasing the number of flash channels and chips with the rapidly shrinking process and manufacturing technology [22, 25, 41, 46].

The flash translation layer (FTL) is the core component of managing flash memory in SSDs, including address translation, garbage collection (GC), and wear leveling [20, 66]. The FTL maintains metadata structures for different functions such as address translation and valid page tracking, and caches them in the in-device DRAM (SSD DRAM) for improved performance [7, 12, 25].

Among these data structures, the address mapping table has the largest memory footprint. In general, the address mapping table can be categorized in three types: page-level mapping, block-level mapping, and hybrid mapping. Modern SSDs usually use the page-level mapping, as it offers the best performance for the flash page lookup, and incurs minimal GC overhead, in comparison with the other two mapping schemes [20, 66]. However, the page-level mapping table size is large, as it stores the entry for the LPA-to-PPA address translation for each flash page.

The address mapping table significantly affects the performance of SSDs, as it not only determines the efficiency of indexing flash pages, but also affects the utilization of SSD DRAM. Moreover, due to the limitations of the cost and power budget in SSD controllers, it is challenging for SSD vendors to scale the in-device DRAM capacity [12, 41]. This challenge becomes even worse with the increasing flash memory capacity in an SSD, as larger capacity usually requires a larger address mapping table for indexing.

To improve the address mapping and translation for SSDs, various optimization schemes have been developed [9, 25, 29, 38, 39, 66]. However, most of them were developed based on human-driven heuristics [25], and cannot capture dynamic data access patterns at runtime. Employing more semantic knowledge into the FTL, such as GraphSSD [44], can improve the data indexing and address translation, however, it is application specific and complicates the

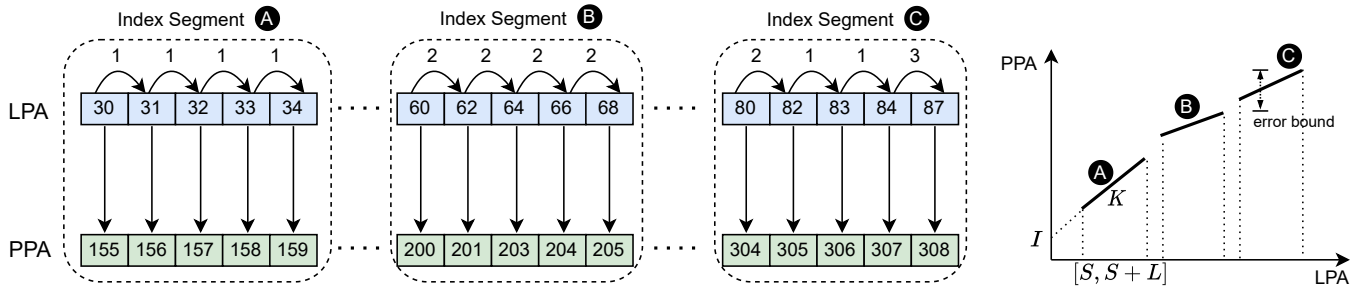


Figure 1: An illustrative example of learning LPA-PPA mappings using piecewise linear regression in LeaFTL. It can learn various patterns of LPA-PPA mappings with guaranteed error bound. Each learned index segment can be represented with (S, L, K, I), where $[S, S+L]$ denotes the interval of LPAs, K is the slope, and I is the intercept of the index segment.

management of address mappings [7], which does not scale for the development of generic SSDs. In this work, we do not expect that we can obtain application semantics from the host and the SSD controller. Instead, we focus on utilizing simple yet effective machine learning (ML) techniques to automate the address mapping table management in the SSDs, with the capability of learning diverse and dynamic data access patterns.

To this end, we propose a learning-based FTL, named LeaFTL, by utilizing the piecewise linear regression technique to learn the LPA-PPA mappings, and automatically exploiting the data locality of various data access patterns at runtime. Unlike the state-of-the-art page-level mapping, the key idea of LeaFTL is that it can learn the correlation between a set of LPAs and their mapped PPAs, based on which it can build a space-efficient index segment, as presented in A in Figure 1. Since the learned index segment can be simply represented with (S, L, K, I), where $[S, S+L]$ denotes the interval of LPAs, K is the slope of the segment, and I is the intercept of the segment (see the last diagram in Figure 1), each segment will take only 8 bytes (1 byte for S and L, 2 bytes for K, and 4 bytes for I) with our optimizations (see the details in §3). Compared to the on-demand page-level mapping [20], the learned segment reduces the mapping table size by a factor of $m * avg(L)/8$, where m is the size (8 bytes) of each entry in the on-demand page-level mapping table, and $avg(L)$ is the average number of LPA-PPA mappings that can be represented in a learned index segment, $avg(L)$ is 20.3 according to our study of various storage workloads.

Beyond learning contiguous LPA-PPA mappings, LeaFTL also learns different correlation patterns, such as regular and irregular strided data accesses as shown in B and C, respectively. Unlike existing indexing optimizations based on human-driven heuristics, LeaFTL can learn more irregular patterns of LPA-PPA mappings with guaranteed error bound, as shown in C. This enables LeaFTL to further condense the address mapping table. Therefore, given a limited DRAM capacity in the SSD controller, LeaFTL can maximally utilize the DRAM caching and improve the storage performance. For the worst case like random I/O accesses, LeaFTL will transfer the mapping into single-point linear segments ($L = 0$, $K = 0$, and $I = PPA$ in Figure 1), and its memory consumption will be no more than that of the page-level mapping.

With the learned index segments, LeaFTL may occasionally return an inaccurate PPA (i.e., address misprediction), which incurs additional flash accesses until the correct PPA is identified. To overcome this challenge, we develop an error-tolerant mechanism in LeaFTL. For each flash page access, we use the reverse mapping stored in the out-of-band (OOB) metadata of each flash page to verify the correctness of the data access. Since the OOB usually has 64–256 bytes [20, 23], we use it to store the accurate LPAs mapped to the neighbor PPAs. Thus, upon an address misprediction, we use the stored reverse mappings to find the correct PPA, avoiding additional flash accesses. LeaFTL leverages the intrinsic OOB structure to handle address mispredictions and make SSD perfectly-suited for practical learned indexing.

Due to the intrinsic out-of-place write property of SSDs (see §2), the learned index segments will be disrupted by writes and GC, and the segments need to be relearned with new LPA-PPA mappings. To tolerate these disruptions, the learned segments are organized within multiple levels to maintain the temporal order in a log-structured manner: the topmost level has the most recent segments, and the lower level stores older segments. The segments at the same level are sorted without overlapping. If the new segment has a conflict with an existing segment, the old segment will be moved to the lower level. Therefore, LeaFTL can always identify the latest version of the corresponding LPA-PPA mapping in a top level of learned index segments. LeaFTL will compact the learned segments periodically to reduce its memory footprint.

To further maximize the efficiency of LeaFTL, we coordinate its learning procedure with flash block allocation in the SSD. As flash block allocation decides the distribution of mapped PPAs, LeaFTL will allocate consecutive PPAs to contiguous LPAs at its best effort, for increasing the possibility of learning a space-efficient index segment. Similar to existing page-level mapping [20, 23], LeaFTL stores the learned index segments in flash blocks for recovery. Overall, we make the following contributions:

- We present a learning-based FTL, it can learn various data access patterns and turn them into index segments for reducing the storage cost of the mapping table.
- We develop an error-tolerant address translation mechanism to handle address mispredictions caused by the learned indexes, with minimal extra flash accesses.
- We preserve the core FTL functions, and enable the coordination between the learning procedure of the address mapping table

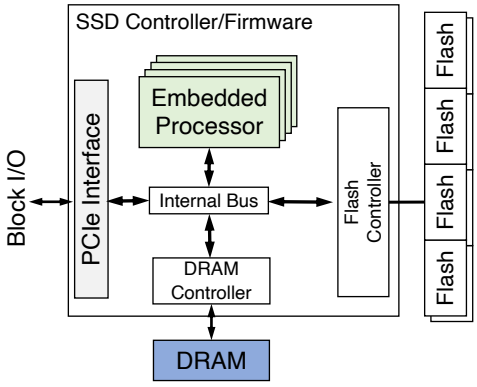


Figure 2: The internal system architecture of SSDs.

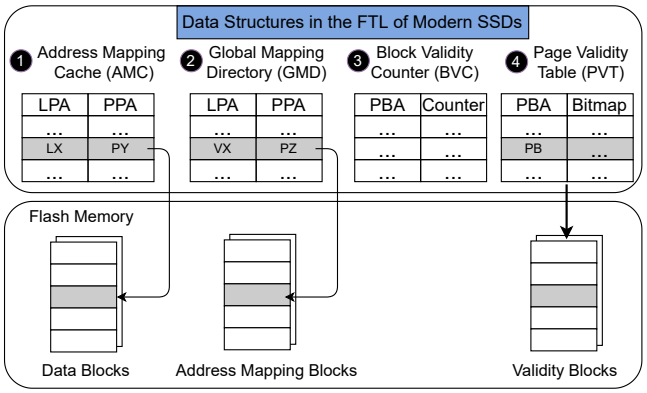


Figure 3: The common data structures in the FTL of SSDs.

with the flash block allocation and GC to maximize the efficiency of the learned FTL.

- We manage the learned segments in an optimized log-structured manner, and enable compaction to further improve the space efficiency for the address mapping.

We implement LeaFTL with a validated SSD simulator WiscSim [27] and evaluate its efficiency with a variety of popular storage workloads. We also develop a system prototype with a real 1TB open-channel SSD to verify the functions of LeaFTL and validate its efficiency with real data-intensive applications, such as the key-value store and transactional database. Our evaluation with the real SSD shows similar benefits as that of the SSD simulator implementation. We demonstrate that LeaFTL reduces the storage cost of the address mapping in the FTL by 2.9× on average. The saved memory space benefits the utilization of the precious SSD DRAM, and further improves the storage performance by 1.4× on average. We also show that LeaFTL does not affect the SSD lifetime, and its learning procedure introduces negligible performance overhead to the storage processor in the SSD controllers. The codebase of LeaFTL is available at https://github.com/platformxlab/LeaFTL.

2 BACKGROUND AND MOTIVATION

Flash-Based Solid-State Drive. An SSD has three major parts (see Figure 2): a set of flash memory packages, an SSD controller with embedded processors, and a set of flash controllers. With the nature of NAND Flash, when a free page is written, the page cannot be written again until that page is erased. However, erase operation is performed only at a block granularity. As the erase operation is expensive, writes are issued to free flash pages erased in advance (i.e., out-of-place write). GC will be performed to clean the stale data. As each flash block has limited endurance, it is important for them to age uniformly (i.e., wear leveling). SSDs have a logical-to-physical address mapping table to index flash pages. All these functions are managed by the FTL in the SSD firmware.

Modern SSD controllers have general-purpose embedded processors (e.g., ARM processors). The processors help with issuing I/O requests, translating LPAs to PPAs, and handling GC and wear-leveling. SSDs also have limited DRAM capacities to cache the mapping table and the application data.

Address Mapping Table in the FTL. The address mapping table in FTL generally has three types: page-level mapping, block-level mapping, and hybrid mapping. The page-level mapping enables direct LPA-PPA mapping for fast lookup. However, each entry usually takes 8 bytes (4 bytes for LPA, 4 bytes for PPA), and the entire mapping table requires large storage space. The block-level mapping significantly reduces the mapping table size. However, it introduces additional overhead for the page lookup in the flash block. The hybrid mapping takes advantages of both page-level and block-level mapping. It uses log blocks to store new writes, and index them with the page-level mapping. The log blocks will be moved into data blocks that are indexed with block-level mapping. This incurs significant GC overhead. Therefore, modern SSDs commonly use the page-level mapping scheme.

Metadata Structures for Flash Management. The FTL usually employs four metadata structures (see Figure 3): (1) the address mapping cache (❶ AMC) for caching the address mapping table in the SSD DRAM; (2) the global mapping directory (❷ GMD) for tracking the locations of the address mapping table pages in the SSD; (3) the block validity counter (❸ BVC) for tracking the number of valid pages for each flash block for assisting the GC in the SSD; and (4) the page validity table (❹ PVT), which uses bitmaps to track the valid pages in each flash block. During the GC, the FTL will check the ❸ BVC to select candidate flash blocks, and migrate their valid pages to free flash blocks. After that, it will erase these selected flash blocks, and mark them as free blocks.

Limited DRAM Capacity in SSD Controllers. It is hard to provision large DRAM inside SSD controllers, due to their hardware constraints and limited budgets for power and hardware cost [12, 41, 60]. Thus, SSD controllers often use on-demand caching to maintain the recently accessed metadata and data in the SSD DRAM.

Among all the metadata structures, the address mapping table has the largest memory footprint. As discussed, ❶ AMC caches the recently accessed mapping table entries. If a mapping entry is not cached, the FTL will locate the corresponding address mapping table pages stored in the flash blocks, and place the mapping entry in the ❶ AMC. As we scale the SSD capacity, the DRAM challenge will become even worse. To overcome this challenge, various optimizations on the mapping table have been proposed [9, 25, 29, 31, 38, 39] to improve the utilization of the SSD DRAM. However, most of

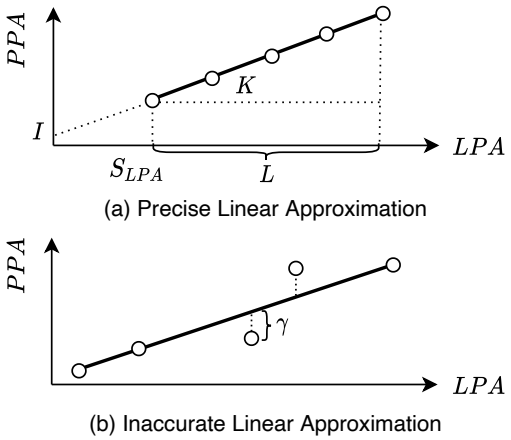


Figure 4: Visualization of learned index segments.

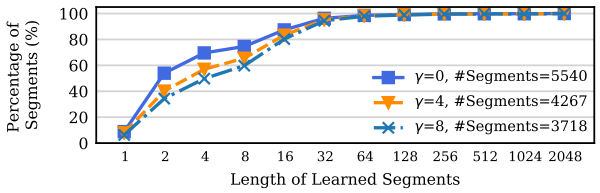
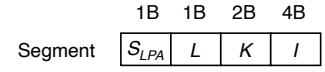


Figure 5: Aggregated distribution of learned segments.



Type	LPAs	PPAs	Index Segment
Accurate	[0, 1, 2, 3]	[32, 33, 34, 35]	0 | 3 | 1.00 | 32
Approximate	[0, 1, 4, 5]	[64, 65, 66, 67]	0 | 5 | 0.56 | 64

Figure 6: Types of learned segments in LeaFTL.

them cannot automatically capture diverse data access patterns at runtime, leaving a large room for improvement.

3 DESIGN AND IMPLEMENTATION

To develop LeaFTL in the SSD controller, we have to overcome the following research challenges.

- LeaFTL should be able to automatically capture diverse data access patterns, and generate memory-efficient address mapping (§3.1, §3.2, §3.3, and §3.4).
- LeaFTL may incur address mispredictions, which could incur additional flash accesses. LeaFTL should be tolerant of errors and have low misprediction penalty (§3.5).
- LeaFTL should work coordinately with other core FTL functions that include GC and wear leveling (§3.6).
- LeaFTL should be lightweight and not incur much extra overhead to storage operations (§3.7, §3.8 and §3.9).

3.1 Key Ideas of LeaFTL

Instead of using the space-consuming one-to-one mapping in the page-level mapping, the key idea of LeaFTL is to exploit learning techniques to identify various LPA-PPA mapping patterns and build efficient learned address mapping entries. Modern SSD controllers usually have a data buffer for grouping writes and write the large data chunk at once for exploiting the internal flash parallelisms. LeaFTL utilizes this data buffer to collect LPA-to-PPA mappings for learning index segments for free, and does not introduce extra data collection overhead (see the details in §3.3).

As shown in Figure 4 (a), the PPA of an LPA can be obtained with the expression: $PPA = f(LPA) = \lceil K * LPA + I \rceil$, $LPA \in [S_{LPA}, S_{LPA} + L]$, where $[S_{LPA}, S_{LPA} + L]$ denotes the interval (L) of LPAs, K is the slope, and I is the intercept. As discussed in §1, each learned index segment can be represented in 8 bytes: 1 byte for S_{LPA} and L, respectively; 2 bytes for K, and 4 bytes for I. The size of S_{LPA} is reduced from 4 bytes to 1 byte with our optimizations on the segment management (see §3.4).

We can relax the linear regression to capture more flash access patterns, which further reduces the learned address mapping table size. As shown in Figure 4 (b), the linear regression can learn a pattern with guaranteed error bound $[-\gamma, \gamma]$. As we increase γ, we can cover more flash access patterns. We applied the relaxed linear regression with different γ values to a variety of storage workloads (see §4.1), our experimental results demonstrate that the number of learned index segments is gradually decreased, as we increase γ. Figure 5 shows that 98.2–99.2% of the learned index segments cover up to 128 LPA-PPA mapping entries, demonstrating the potential advantages of the learning-based approach.

As for random access patterns, LeaFTL will transfer the learned segments into single-point segments. And these linear segments do not require more storage space than the page-level mapping.

3.2 Learned Index Segment

Types of Learned Index Segment. The mapping table of LeaFTL is built with learned index segments. It has two types of segments: accurate and approximate segments, as shown in Figure 6. Both of them are learned with piecewise linear regression technique [64].

As for the accurate index segments, given an LPA, we can precisely get the corresponding PPA with $f(LPA) = \lceil K * LPA + I \rceil$. For example, when the LPA is 2 in Figure 6, we can directly get the PPA value of 34 with $\lceil 1.00 * 2 + 32 \rceil$. In this example, the learned segment has $L = 3$ and it indexes 4 LPA-PPA mappings. If $L = 0$, the learned segment will become a single-point segment, the slope $K = 0$, and we will get its PPA with $PPA = I$.

As for approximate index segments, we use the same formula $f(LPA) = \lceil K * LPA + I \rceil$ to calculate the PPA. However, the returned PPA may not be the exact corresponding PPA. It has an error bound $[-\gamma, \gamma]$ guaranteed by the linear regression, and γ is configurable. For example, given $LPA = 4$ in Figure 6, the value of the PPA is 67, according to the calculation $\lceil 4 * 0.56 + 64 \rceil$. However, the real PPA should be 66. We define this as *address misprediction*. We will

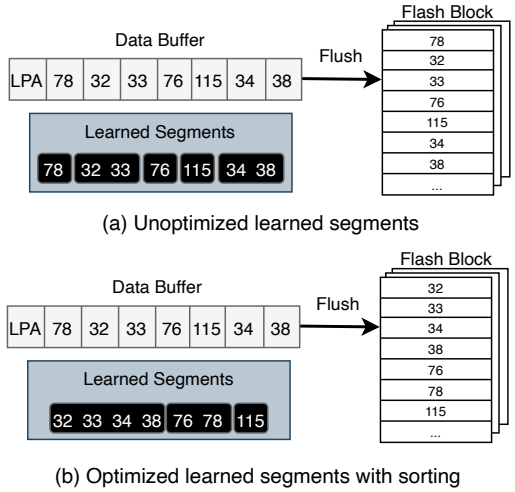


Figure 7: An example of reducing the number of learned segments via exploiting the flash block allocation.

discuss how we handle the address misprediction with reduced miss penalty in §3.5.

Size of Learned Index Segment. As discussed in §3.1, each segment can be expressed in (S_{LPA}, L, K, I). The starting LPA will take 4 bytes. We can further reduce this size by partitioning a range of LPAs into small groups, and each LPA group represents a certain number of contiguous LPAs. Therefore, we can index an LPA with its offset in a corresponding group. In LeaFTL, each group represents 256 contiguous LPAs. Thus, S_{LPA} can be indexed by the offset ($2^8 = 256$) in the group, which takes only 1 byte. We use 256 as the group size, because the length of the learned segments is usually less than 256 (see Figure 5).

Given an LPA, we can get its offset in the group with ($LPA \ mod$ 256). In LeaFTL, we set the L as 1 byte. Thus, each segment can index 256 LPA-PPA mappings. We use a 16-bit floating point to store the value of the slope K. And the intercept I of a segment can be represented in 4 bytes. Therefore, in combination with S_{LPA}, both accurate and approximate segments can be encoded with 8 bytes (see Figure 6), which are memory aligned.

LeaFTL uses the least significant bit of the K to indicate segment types (0 for accurate segments, 1 for approximate segments). This has negligible impact on the address translation accuracy, because $K \in [0, 1]$, which will only affect the tenth digit after decimal point.

3.3 Improve the Learning Efficiency

To further reduce the number of learned segments, LeaFTL performs optimizations to improve its learning efficiency of address mappings by exploiting the flash block allocation in SSD controllers, as shown in Figure 7. Flash pages are usually buffered in the SSD controller and written to flash chips at a flash block granularity, for utilizing the internal bandwidth and avoiding the open-block problem [6, 22, 37, 48]. This allows LeaFTL to learn more space-efficient index segments (i.e., index segments can cover more LPA-PPA mappings) by reordering the flash pages with their LPAs in the data buffer. As shown in Figure 7 (a), LeaFTL learns 5 index segments (78), (32, 33), (76), (115), and (34, 38) with $\gamma = 4$. After sorting the pages in the data buffer shown in Figure 7 (b), LeaFTL generates 3 index segments (32, 33, 34, 38), (76, 78), and (115).

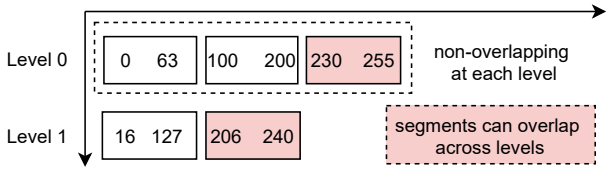


Figure 8: The learned index segments are managed in a log-structured manner in LeaFTL.

To develop the optimized learned segments, LeaFTL sorts the flash pages in ascending order of their LPAs in the data buffer (8MB by default). When pages in the data buffer is flushed to the flash chips, their PPAs are in ascending order. This ensures a monotonic address mapping between LPAs and PPAs, which reduces the number of index segments.

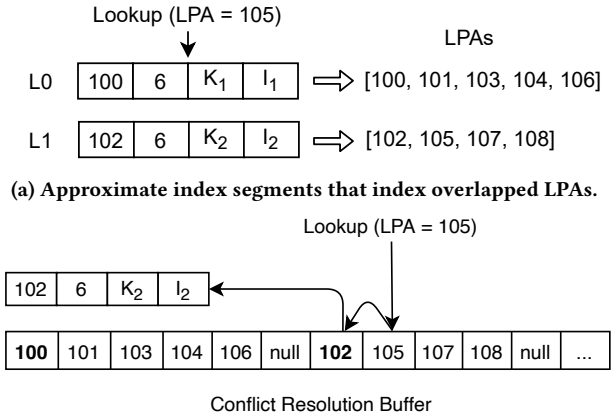
3.4 Manage Learned Index Segments

Upon new data updates or GC in the SSD, the learned index segments need to be updated, due to the intrinsic property (i.e., out-of-place update) of SSDs. Unfortunately, the direct updates to learned index segments are expensive, since we have to relearn the index segments with new PPAs. This relearning procedure not only consumes extra compute cycles, but also involves additional flash accesses, since we have to access the corresponding flash pages to obtain accurate PPAs for some of the LPAs in the index segment being updated. For instance, for in-place update to an approximate segment, it can incur 21 flash accesses on average when relearning. In-place update also breaks the existing LPA-to-PPA mapping patterns, which results in 1.2× additional segments and memory footprint, according to our experiments with various workloads.

To address this challenge, we manage the learned index segments in a log-structured manner, as shown in Figure 8. Therefore, the newly learned index segments will be appended to the log structure (level 0 in Figure 8) and used to index the updated LPA-PPA mappings, while the existing learned segments (level 1 and lower levels in Figure 8) can still serve address translations for LPAs whose mappings have not been updated. Such a structure supports concurrent lookups as enabled in the traditional log-structured merge tree. As we insert the newly learned index segments at the top level of the log-structured tree, this minimizes the impact on other segments.

Log-Structured Mapping Table. The log-structured mapping table has multiple levels to maintain the temporal order of index segments. As discussed, the topmost level has the most recent learned index segments, and the lower level stores the older segments. For the segments on the same level, LeaFTL ensures that they are sorted and do not have overlapped LPAs. This is for fast location of the corresponding learned index segments in each level. For the segments across the levels, they may have overlapped LPAs, due to the nature of the log-structured organization. And the segments with overlapped LPA-PPA mappings will be compacted periodically for space reclamation (see its detailed procedure in §3.7).

Manage Two Types of Index Segments. LeaFTL manages the accurate and approximate index segments in the same log-structured



(a) Approximate index segments that index overlapped LPAs.

(b) Resolve the conflict between approximate segments with CRB

Figure 9: A case study of conflict resolution buffer for approximate learned index segments.

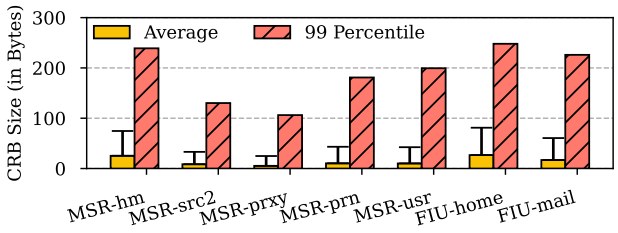


Figure 10: The distribution of CRB sizes for different storage workloads, when we set $\gamma = 4$ in LeaFTL.

mapping table, as they can be encoded in the same format. For each accurate segment, we can directly infer its indexed LPAs with the S_{LPA}, K, and L, since it has a regular pattern. However, for approximate index segments, we only have the knowledge of the starting LPA and the end LPA with $S_{LPA} + L$. Its encoded LPAs cannot be directly inferred from their metadata (S_{LPA}, L, K, I), since they are learned from irregular access patterns and may have mispredictions.

If two approximate segments have overlapping LPA ranges, we could obtain inaccurate PPAs from the learned index segments. As shown in Figure 9 (a), given an LPA with the value 105, we will check the segment at Level 0 and may get an inaccurate PPA. This will also affect the efficiency of the segment compaction, with which we eliminate duplicated entries between segments.

To address this challenge, LeaFTL uses a Conflict Resolution Buffer (CRB) for each LPA group to store the LPAs indexed by each approximate segment. The main purpose of CRB is to help LeaFTL check whether a given LPA belongs to one approximate segment.

The CRB is a nearly-sorted list [10] by the starting LPAs of its approximate segments. To be specific, the CRB ensures the following properties: (1) the LPAs belong to the same approximate segment are stored contiguously; (2) different approximate segments are sorted by their starting LPA, and CRB uses a *null* byte to separate these segments; (3) it does not have redundant LPAs, which means an LPA will appear at most once in the CRB. This is achieved by removing existing same LPAs when we insert new approximate segments into the CRB.

However, if the S_{LPA} of a new approximate segment is the same as any starting LPAs that have been stored in the CRB, LeaFTL will update the S_{LPA} of the old segment with the adjacent LPA. Take Figure 9 (b) as an example, upon a new approximate segment with $S_{LPA} = 100$, we will update the S_{LPA} of the existing segment to 101, and then insert the new segment into the CRB. In this case, LeaFTL will ensure each approximate segment will have its unique S_{LPA}. This will facilitate the approximate LPA-PPA address translation with high accuracy confidence.

Since CRB is nearly sorted, its insertion, deletion, and lookup operations are fast. The CRB is also space efficient, as each LPA (the offset in its corresponding LPA group) will take only one byte, and it guarantees that there are no redundant LPAs. Therefore, the CRB will maximally store 256 LPAs. Our experiments with a variety of storage workloads show that the CRB will take 13.9 bytes on average, as shown in Figure 10.

Given an LPA, in order to identify which approximate index segment it belongs to, LeaFTL will check the CRB with binary search. Once the LPA is found, LeaFTL will search to its left until identifying the S_{LPA}, and this S_{LPA} will be the starting LPA of the corresponding approximate segment, as shown in Figure 9 (b). Therefore, CRB can assist LeaFTL to resolve the LPA lookups.

3.5 Handle Address Misprediction

As discussed in §3.2, the mapping table entries encoded with approximate segments may occasionally incur mispredictions and return an approximated PPA. These approximate segments have a guaranteed error bound $[-\gamma, \gamma]$, where γ is a constant value that can be specified in the linear regression algorithm. To verify the correctness of the address translation, a simple method is to access the flash page with the predicted PPA, and use the reverse mapping (its corresponding LPA) stored in the OOB metadata of the flash page to check whether the LPA matches or not. In this case, upon a PPA misprediction, we need $\log(\gamma)$ flash accesses on average to identify the correct PPA.

To avoid extra flash accesses for address mispredictions, LeaFTL leverages the OOB of the flash page to store the reverse mappings of its neighbor PPAs. This is developed based on the insight that: with a $PPA_{learned}$ obtained from an approximate segment, its error bound $[-\gamma, \gamma]$ guarantees that the correct PPA is in the range of $[PPA_{learned} - \gamma, PPA_{learned} + \gamma]$, as discussed in Figure 4 (b). Thus, upon a misprediction, LeaFTL will read the flash page with $PPA_{learned}$, and use its OOB to find the correct PPA. In this case, LeaFTL ensures that it will incur only one extra flash access for address mispredictions.

This is a feasible approach, as the OOB size is usually 128–256 bytes in modern SSDs. As each LPA takes 4 bytes, we can store 32–64 reverse mapping entries in the OOB. We show the OOB organization of LeaFTL in Figure 11. For the flash page PPA_X, the first $2\gamma + 1$ entries in its OOB correspond to the LPAs for the flash pages $[PPA_X - \gamma, PPA_X + \gamma]$. For the flash pages at the beginning and end of a flash block, we may not be able to obtain the reverse mapping of their neighbor PPAs. We place the *null* bytes in the corresponding entry of the OOB.

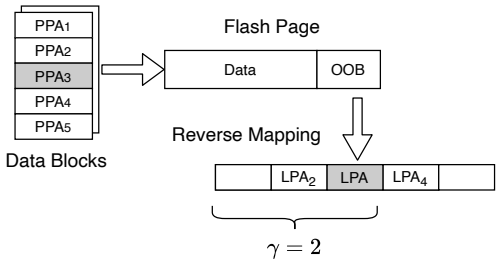


Figure 11: The out-of-band (OOB) metadata organization. It stores the reverse mapping for its neighbor PPAs.

3.6 Preserve Other Core FTL Functions

LeaFTL preserves the core functions such as GC and wear leveling in an FTL. It follows the same GC and wear leveling policies in modern SSDs. When the number of free blocks in an SSD is below a threshold (usually 15-40% of the total flash blocks), the SSD controller will trigger the GC execution. LeaFTL employs the greedy algorithm [5] to select the candidate blocks which have the minimal number of valid pages, for reducing the data movement overhead at GC. As the GC move the valid pages from the candidate blocks to the free blocks, LeaFTL places these valid pages into the DRAM buffer, sort them by their LPAs, and learn a new index segment. The learning procedure is the same as we build index segments for new flash writes/updates. Thus, the address mapping of the valid pages is updated after the GC.

LeaFTL also ensures all the flash blocks age at the same rate (i.e., wear leveling). It uses the throttling and swapping mechanism developed in existing GC, in which the cold data blocks (i.e., blocks not frequently accessed) will be migrated to hot blocks (i.e., blocks that experience more wear). LeaFTL will learn new indexes for these swapped blocks and insert them into the mapping table to update their address mappings.

3.7 LeaFTL Operations

Now we describe the LeaFTL operations, including segment creation, insert/update, LPA lookup, and compaction. We discuss their procedures, and use examples to illustrate each of them, respectively. We present their detailed procedures in Algorithm 1 and 2.

Creation of Learned Segments. Once the data buffer of the SSD controller is filled, LeaFTL takes the LPAs and PPAs of the flash pages in the buffer as the input. It sorts the LPA-PPA mappings by reordering the flash pages with their LPAs (see §3.3), and uses greedy piecewise linear regression [64] to learn the index segment.

Insert/Update of Learned Segments. When we insert or update a new learned index segment, we will place it in the topmost level of the log-structured mapping table. Since each level of the mapping table is sorted, we can quickly identify its insert location via a binary search (line 2 in Algorithm 1). If the new segment is approximate, LeaFTL will update the CRB for future lookups (line 4-7 in Algorithm 1). After that, LeaFTL will check whether the new segment overlaps with existing segments. If yes, LeaFTL will identify the overlapped LPAs. The overlap detection is performed by the comparison between the LPA range of the new segment and

ALGORITHM 1: LeaFTL operations

```
   Input: groups ← LeaFTL group partitions
   // Insert/Update Segment in the LeaFTL
1  Function seg_update(segment, level):
2      seg_pos = binary_search(level, segment.S_LPA)
3      level.insert(segment, seg_pos)
4      if not segment.accurate then
5          Insert LPAs into CRB and remove redundant LPAs
6          if segment.S_LPA exists in CRB then
7              Update the S_LPA of the old segment
8      victim_segments ← All segments that overlap the segment
         starting with seg_pos
9      foreach victim ∈ victim_segments do
10         seg_merge(segment, victim)
             // if marked as removable by seg_merge()
11         if victim.L = −1 then
12             level.remove(victim)
13         if segment.overlaps(victim) then
14             Pop victim to the next level
15             if victim has overlaps in the next level then
16                 Create level for victim to avoid recursion
   // Lookup LPA in the LeaFTL
17 Function lookup(lpa):
18     foreach level ∈ groups[lpa mod 256] do
19         seg_pos = binary_search(level, lpa)
20         segment = level.get_segment(seg_pos)
21         if has_lpa(segment, lpa) then
22             return segment.translatePPA(lpa)
   // LeaFTL Compaction
23 Function seg_compact():
24     foreach group ∈ groups do
25         foreach upper_level, lower_level ∈ group do
26             foreach segment ∈ upper_level do
27                 seg_update(segment, lower_level)
28             if upper_level is empty then
29                 group.remove(upper_level)
```

$[S_{LPA}, S_{LPA}+L]$ of the adjacent segments. We group these overlapping segments as a list of victim segments (line 8 in Algorithm 1). LeaFTL will merge segments to remove outdated LPAs (line 10 in Algorithm 1 and line 14-25 in Algorithm 2).

To fulfill the segment merge, LeaFTL will use the S_{LPA}, L, and K to reconstruct the list of the encoded LPAs in the victim segment. And it will create a bitmap to index these encoded LPAs (line 6-13 in Algorithm 2). Given an accurate segment with $S_{LPA} = 100, K = 0.5, L = 6$, we can infer that its encoded LPAs are $[100, 102, 104, 106]$. We can transfer the LPA list to the bitmap $[1010101]$. If the victim segment is an approximate segment, LeaFTL will leverage the S_{LPA}, L, and the LPAs stored in the CRB to reconstruct the encoded LPAs. Afterwards, LeaFTL will conduct a comparison between the bitmaps to identify the overlapped LPAs (line 15-19 in Algorithm 2).

During the segment merge, LeaFTL will update the S_{LPA} and L of the old segments accordingly, remove the outdated LPAs from CRB for approximate segments. Note that we do not update the K and I for the victim segments during the merge.

After the merge, (1) if the victim segment does not contain any valid LPA (L is negative), it will be removed from the mapping table (line 11-12 in Algorithm 1). (2) If the victim segment has

ALGORITHM 2: Segment Merge

```
   // Check if Segment Contains LPA
1  Function has_lpa(seg, lpa):
2      acc ← seg.accurate
3      if lpa ∉ [seg.S_LPA, seg.S_LPA + seg.L] or
         (not acc & check(CRB) failed) or
         (acc & (lpa − seg.S_LPA) mod ⌈1/seg.K⌉ ≠ 0) then
4          return False
5      return True
   // Convert Segment into a Temporary Bitmap
6  Function get_bitmap(seg, start, end):
7      bm ← bitmap of length (end − start + 1)
8      foreach lpa ∈ [start, end] do
9          if has_lpa(seg, lpa) then
10             bm[lpa − start] = 1
11         else
12             bm[lpa − start] = 0
13     return bm
   // Merge a New Segment with an Old Segment
14 Function seg_merge(new, old):
15     start ← min(new.S_LPA, old.S_LPA)
16     end ← max(new.S_LPA + new.L, old.S_LPA + old.L)
17     bm_new ← get_bitmap(new, start, end)
18     bm_old ← get_bitmap(old, start, end)
19     bm_old ← bm_old & ¬bm_new
20     first, last ← the first and last valid bit of bm_old
21     old.S_LPA, old.L ← first + start, last − first
22     if no valid bits in old then
23         old.L ← −1                    // mark it as removable
24     if not old.accurate then
25         Remove outdated LPAs in CRB
```

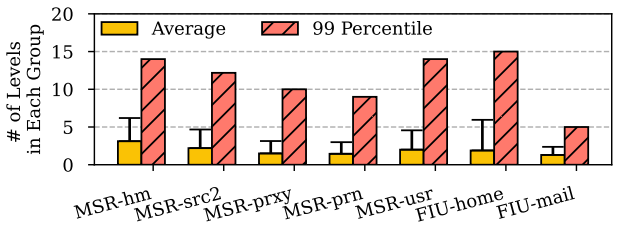


Figure 12: A study of the number of levels in the log-structured mapping table for different storage workloads.

valid LPAs but their range still overlaps with the new segment, the victim segment will be moved to the next level in the log-structured mapping table (line 13-16 in Algorithm 1). To avoid recursive updates across the levels, we create a new level for the victim segment if it also overlaps with segments in the next level. According to our study of diverse workloads, this will not create many levels in the mapping table (see Figure 12). (3) If the victim segment has valid LPAs and they do not overlap with the new segment, we do not need to perform further operations. This is because the victim segment is updated with new S_{LPA} and L during segment merge (line 20-25 in Algorithm 2), and the new segment insertion keeps each level sorted (line 3 in Algorithm 1).

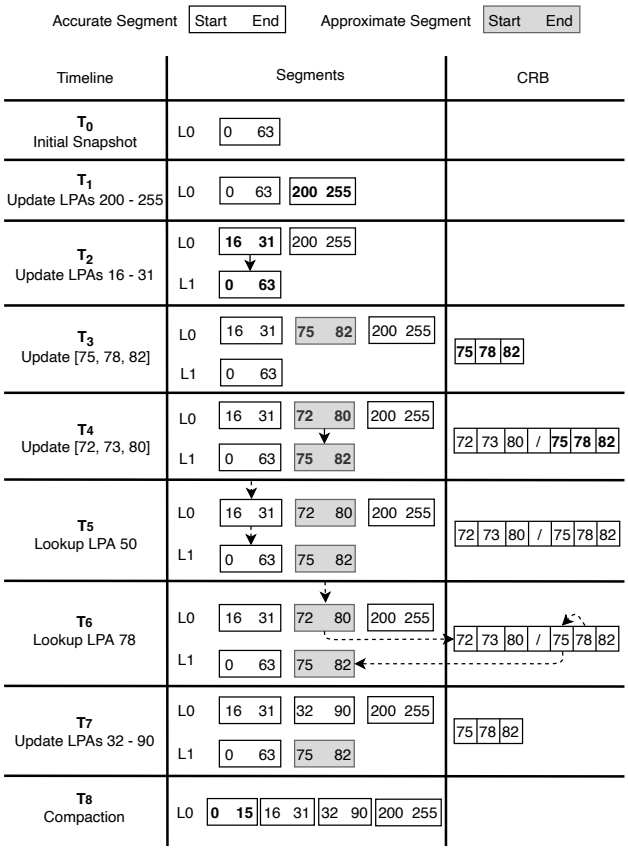


Figure 13: Examples that involve update/insert, lookup, and compaction operations in LeaFTL.

To facilitate our discussion, we present a few examples in Figure 13. At the initial stage, the mapping table has one segment that indexes the LPA range [0, 63]. At T_1, the new segment [200, 255] is directly inserted into the topmost level, as it does not overlap with existing segments. At T_2, we insert a new segment [16, 31] that has overlaps with the old segment [0, 63], LeaFTL conducts the segment merge procedure. After that, the old segment still has valid LPAs. Thus, it moves to level 1. At T_3 and T_4, we insert two approximate segments [75, 82] and [72, 80], LeaFTL will also insert their encoded LPAs into the CRB. The segment [75, 82] will be moved to the next level as it overlaps with the new segment [72, 80].

LPA Lookup. LeaFTL conducts an LPA lookup from the topmost level of the mapping table with binary searches (line 19 in Algorithm 1). We will check whether the LPA is represented by the matched segment (line 21 in Algorithm 1, line 1-5 in Algorithm 2). If the $LPA \in [S_{LPA}, S_{LPA} + L]$ of the segment, LeaFTL will check the least bit of its K. If the least bit of K is 0, it is an accurate segment, and LeaFTL will use $f(LPA) = \lceil K * LPA + I \rceil$ to get the accurate PPA (see §3.2). Otherwise, it is an approximate segment. LeaFTL will check the CRB to identify the S_{LPA} of the segment, following the approach described in Figure 9 and §3.4. LeaFTL will use the same $f(LPA)$ formula to obtain the PPA. If the LPA is not found in the top level of the mapping table, LeaFTL will search the lower levels until a segment is identified.

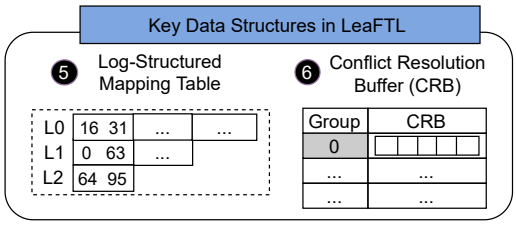


Figure 14: Key data structures used in LeaFTL.

We use Figure 13 to illustrate the lookup procedure. At T_5, we conduct the address translation for $LPA = 50$. However, none of the segments in the level 0 covers this LPA, LeaFTL will continue the search in the level 1 and find the accurate segment [0, 63]. At T_6, we do the address translation for $LPA = 78$. LeaFTL finds that the LPA 78 is in the LPA range of the segment [72, 80]. Since this is an approximate segment, LeaFTL checks the CRB and finds this LPA is actually indexed by the segment [75, 82].

With the PPA, LeaFTL will read the corresponding flash page and use the reversed mapping (its corresponding LPA) in its OOB to verify the correctness of the address translation. Upon mispredictions, we will use the approach discussed in §3.5 to handle it.

Segment Compaction. The purpose of the compaction is to merge segments with overlapped LPAs across different levels, which further saves memory space. LeaFTL will iteratively move the upper-level segments into the lower level, until the mapping table is fully compacted (line 27 in Algorithm 1). When an approximate segment is removed, its corresponding CRB entries will also be deleted. As shown in T_7 of Figure 13, we insert a new segment [32, 90] which fully covers the LPA range of the segment [72, 80]. After merge, LeaFTL removes the old segment [72, 80]. However, some segments in the level 0 still overlap with the segments in the level 1. After T_8, LeaFTL will remove outdated segments and LPAs.

LeaFTL performs segment compaction after each 1 million writes by default. According to our experiments with various storage workloads, the segment compaction of the entire mapping table will take 4.1 milliseconds (the time of 20-40 flash writes) on average. Consider the low frequency (i.e., once per 1 million writes), the compaction incurs trivial performance overhead to storage operations.

3.8 Put It All Together

LeaFTL is compatible with existing FTL implementations. As shown in Figure 14, it uses the log-structured mapping table (❺) to replace the address mapping cache (❶ in Figure 3), and employs CRB (❻) for assisting the address translation of approximate segments. The CRB requires trivial storage space in the SSD DRAM (see Figure 10).

Read Operation. For a read request, LeaFTL will first check the data cache. For a cache hit, LeaFTL serves the read request with the cached flash page. Otherwise, LeaFTL will perform address translation with ❺ (see §3.7). If there is a misprediction of PPA, LeaFTL checks the OOB of the mispredicted flash page, read the correct page (§3.5), and updates the data cache with the page.

Write Operation. For a write request, LeaFTL buffers it in the data cache. Once the buffered writes reach the size of a flash block, LeaFTL will allocate a free block. It will sort the writes in the buffer based on their LPAs, and learn new index segments with the PPAs of the allocated flash block. This enables LeaFTL to group more LPA-PPA mappings in the same index segment. After that, LeaFTL will insert the new index segment in the mapping table, and flush the buffered data to the flash blocks. For those writes, LeaFTL will also check whether their LPAs exist in the mapping table. If yes, LeaFTL will update their corresponding entries in ❸ BVC and ❹ PVT to indicate that they become invalid and can be garbage collected in the future. Otherwise, the new learned segments will have their LPA-PPA mappings for future address translations.

LeaFTL caches the mapping table in SSD DRAM for fast lookup. The table will also be stored in the flash blocks. LeaFTL utilizes the existing ❷ GMD to index the translation pages. If a segment is not found in the cached mapping table, LeaFTL will fetch it from the translation blocks and place it in the cached mapping table.

Crash Consistency and Recovery. Upon system crashes or power failures, LeaFTL guarantees the crash consistency of learned indexes. In order to ensure the data durability of DRAM buffer in SSD controllers, modern SSDs today have employed battery-backed DRAM and power loss protection mechanisms [1, 2]. With battery-backed DRAM, LeaFTL has sufficient time to persist the up-to-date mapping table to the flash blocks and record their PPAs in the GMD (❷ in Figure 3). During the data recovery, LeaFTL reads the GMD to locate its mapping table and place it into the DRAM.

Without battery-backed DRAM, LeaFTL periodically flushes the learned mapping table and the Block Validity Counter (❸ BVC in Figure 3) into the flash blocks. When GC is triggered, LeaFTL also flushes the updated mapping table and BVC into the flash blocks. Upon crashes, LeaFTL will scan all the flash blocks at the channel-level parallelism, and reconstruct an up-to-date BVC. LeaFTL is able to identify the flash blocks allocated since the last mapping table flush, by comparing the up-to-date BVC with the stored BVC in the SSD. Therefore, LeaFTL only needs to relearn the index segments for these recently allocated flash blocks and add them into the mapping table (see §3.4).

3.9 Implementation Details

SSD Simulator. We implement LeaFTL based on a trace-driven simulator WiscSim [27], which has provided an event simulation environment for the end-to-end performance analysis of SSDs. We extend WiscSim by implementing an LRU-based read-write cache. LeaFTL also preserves the functions of existing FTL, such as GC and wear-leveling. To support the learned indexing, LeaFTL employs a simple linear regression algorithm [65], which incurs negligible computation overhead with modern storage processors (see §4.5). The error bound γ for learned segments is configurable, and we set it to 0 by default in LeaFTL.

SSD Prototype. We also develop a real system prototype with an open-channel SSD to validate the functions and efficiency of LeaFTL. The SSD has 1TB storage capacity with 16 KB flash page size. It has 16 channels, each channel has 16K flash blocks, and each flash block has 256 pages. It enables developers to implement their own FTL in the host by providing basic I/O commands such as read, write, and erase. We implement LeaFTL with 4,016 lines of code using C programming language with the SDK library of the device.

Table 1: SSD configurations in our simulator.

Parameter	Value	Parameter	Value
Capacity	2TB	#Channels	16
Page size	4KB	OOB size	128B
DRAM size	1GB	Pages/block	256
Read latency	20μs	Write latency	200μs
Erase	1.5 millisecs	Overprovisioning ratio	20%

Table 2: Real workloads used in our real SSD evaluation.

Workload	Description
OLTP [59]	Transactional benchmark in the FileBench.
CompFlow (CompF) [59]	File accesses in a computation flow.
TPCC [13]	Online transaction queries in warehouses.
AuctionMark (AMark) [13]	Activity queries in an auction site.
SEATS [13]	Airline ticketing system queries.

4 EVALUATION

Our evaluation shows that: (1) LeaFTL significantly reduces the address mapping table size, and the saved memory brings performance benefits (§4.2); (2) the benefits of LeaFTL are validated on a real SSD device (§4.3); (3) LeaFTL can achieve additional memory savings and performance benefits with larger error-tolerance, and it demonstrate generality for different SSD configurations (§4.4); (4) Its learning procedure does not introduce much extra overhead to the SSD controller (§4.5); (5) It has minimal negative impact on the SSD lifetime (§4.6).

4.1 Experiment Setup

We examine the efficiency of LeaFTL with both the SSD simulator and real SSD prototype. As for the evaluation with the SSD simulator, we configure a 2TB SSD with 4KB flash pages and 1GB DRAM in the SSD controller. We list the core SSD parameters in Table 1. For other parameters, we use the default setting in the WiscSim. We use a variety of storage workloads that include the block I/O traces from enterprise servers from Microsoft Research Cambridge [45] and workload traces from computers at FIU [16]. As for the evaluation with the real SSD prototype (see §3.9), we validate the benefits of LeaFTL using a set of real-world file system benchmarks and data intensive applications as shown in Table 2. Before we measure the performance, we run a set of workloads consisting of various real-world and synthetic storage workload traces to warm up the SSD and make sure the GC will be executed during the experiments.

We compare LeaFTL with state-of-the-art page-level mapping schemes described as follows [1].

- **DFTL (Demand-based FTL)** [20]: it uses a page-level mapping scheme, and caches the most recently used address translation entries in the SSD DRAM.
- **SFTL (Spatial-locality-aware FTL)** [25]: it is a page-level mapping that exploits the spatial locality and strictly sequential access patterns of workloads to condense mapping table entries.

[1]We do not compare LeaFTL with block-level and hybrid-level mappings, as they perform dramatically worse than the page-level mapping [20, 25].

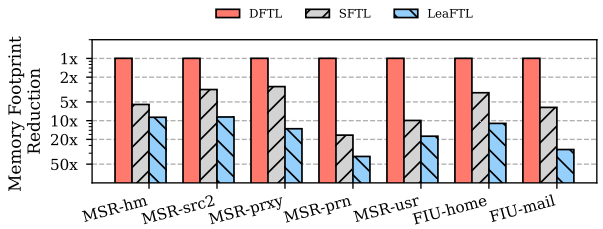
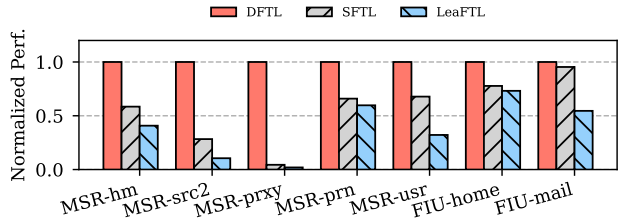
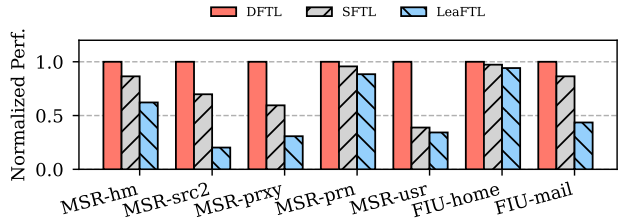


Figure 15: The reduction on the mapping table size of LeaFTL, in comparison with DFTL and SFTL.



(a) SSD performance when using its DRAM mainly for the address mapping table (lower is better).



(b) SSD performance when using its DRAM partially (up to 80%) for the address mapping table (lower is better).

Figure 16: Performance improvement with LeaFTL.

4.2 Memory Saving and Performance

We first evaluate the benefits of LeaFTL on the memory saving and storage performance with the SSD simulator. As shown in Figure 15, LeaFTL reduces the mapping table size by 7.5–37.7×, compared to the page-level mapping scheme DFTL. This is because LeaFTL can group a set of page-level mapping entries into an 8-byte segment. In comparison with SFTL, LeaFTL achieves up to 5.3× (2.9× on average) reduction on the address mapping table for different storage workloads, when we set its $\gamma = 0$ (i.e., the learned segments are 100% accurate). This is because LeaFTL captures more LPA-PPA mapping patterns.

We now evaluate the performance benefit of LeaFTL from its saved memory space. We evaluate LeaFTL with two experimental settings: (1) the SSD DRAM is mainly used (as much as possible) for the mapping table; (2) the SSD DRAM is partially used for the mapping table, in which we ensure at least 20% of the DRAM will be used for the data caching.

In the first setting, DRAM is almost used for mapping table in DFTL. As shown in Figure 16 (a), LeaFTL reduces the storage access latency by 1.6× on average (up to 2.7×), compared to SFTL. This is because LeaFTL saves more memory from the mapping table than SFTL. SFTL slightly outperforms DFTL, because it reduces the

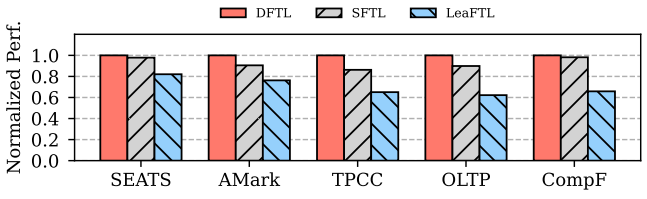


Figure 17: Performance on the real SSD prototype.

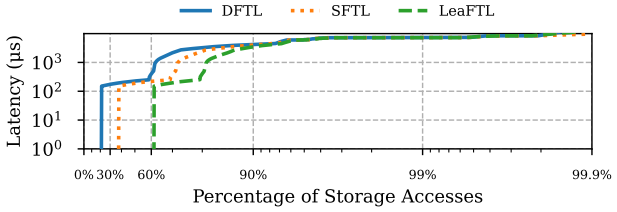


Figure 18: The latency distribution of storage accesses when running OLTP workload on the real SSD prototype.

mapping table size by compressing mapping entries with grouping strictly sequential data accesses. In the second setting, as shown in Figure 16 (b), LeaFTL obtains 1.4× (up to 3.4×) and 1.6× (up to 4.9×) performance speedup, compared to SFTL and DFTL, respectively.

4.3 Benefits on the Real SSD Prototype

We validate the benefits of LeaFTL on the real SSD prototype with real workloads (see Table 2). They include filesystem benchmark suite FileBench [59], and transactional database workloads from BenchBase [13, 61]. All these workloads run on the ext4 file system. With FileBench, we run OLTP and CompFlow (CompF) workloads to read/write 10GB files. With BenchBase, we run TPCC, AuctionMark (AMark), and SEATS workloads on MySQL, and their database sizes are 10–30GB. These database workloads will generate 37–230GB read traffic and 26–59GB write traffic to the SSD. We allocate 256MB DRAM to host the mapping table (for different DRAM sizes, see our sensitivity analysis in §4.4).

We present the performance benefit of LeaFTL in Figure 17. Across all workloads, LeaFTL obtains 1.4× performance speedup on average (up to 1.5×), compared to SFTL and DFTL. Similar to our evaluation with the SSD simulator implementation, the performance benefit of LeaFTL comes from the memory saving from the address mapping table. And LeaFTL demonstrates comparable performance improvement on real SSD devices, in comparison with the SSD simulator in §4.2. We also show the latency distribution of storage accesses in Figure 18, when running the OLTP workload on the real SSD prototype. In comparison with existing FTL schemes, LeaFTL does not increase the tail latency of storage accesses. And the higher cache hit ratio of LeaFTL brings latency reduction for many storage accesses.

4.4 Sensitivity Analysis

Vary the value of γ. As we increase the value of γ from 0 to 16, the size of the learned mapping table is reduced, as shown in Figure 19. LeaFTL achieves 1.3× reduction on average (1.2× on the real SSD) with $\gamma = 16$, compared to that of $\gamma = 0$. The saved

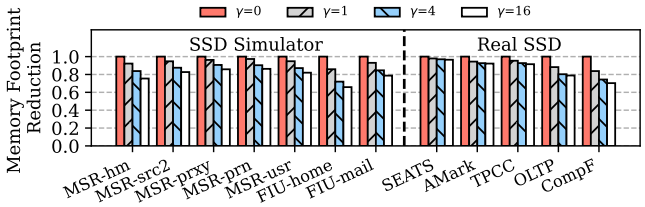


Figure 19: The reduction of the mapping table size of LeaFTL with different γ (lower is better).

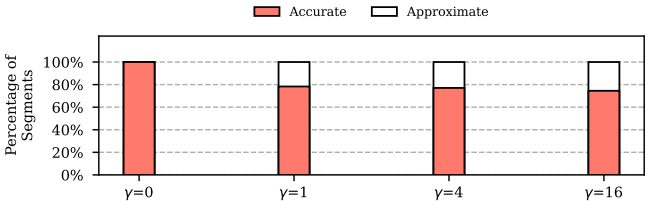


Figure 20: The distribution of learned segments.

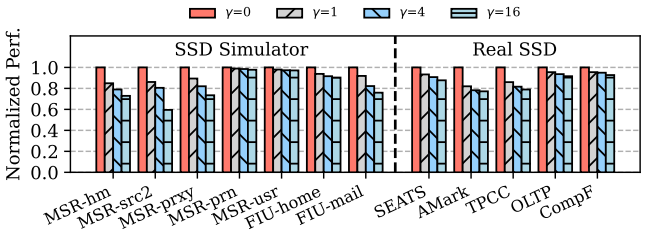


Figure 21: Performance with various γ (lower is better).

memory with a larger γ is achieved by learning a wider range of LPAs into approximate segments. To further understand this, we profile the distribution of segments learned by LeaFTL with different values of γ, as shown in Figure 20. When $\gamma = 0$, all the segments are accurate. When $\gamma = 16$, 26.5% of the learned segments are approximate on average, and LeaFTL delivers 1.3× improvement on storage performance (1.2× with workloads on the real SSD), in comparison with the case of $\gamma = 0$ (see Figure 21).

Vary the SSD DRAM capacity. We now conduct the sensitivity analysis of SSD DRAM by varying its capacity from 256MB to 1GB on the real SSD prototype. As shown in Figure 22 (a), LeaFTL always outperforms DFTL and SFTL as we vary the SSD DRAM capacity. As we increase the DRAM capacity, the storage workloads are still bottlenecked by the available memory space for the data caching. LeaFTL can learn various data access patterns and significantly reduce the address mapping table size, the saved memory further benefits data caching.

Vary the flash page size. In this experiment, we fix the number of flash pages, and vary the flash page size from 4KB to 16KB in the SSD simulator, as SSD vendors usually use larger flash pages for increased SSD capacity. We use the simulator for this study, since the flash page size of the real SSD is fixed. As shown in Figure 22 (b), LeaFTL always performs the best in comparison with DFTL and SFTL. As we increase the flash page size to 16KB, we can cache less number of flash pages with limited DRAM capacity. Thus, LeaFTL

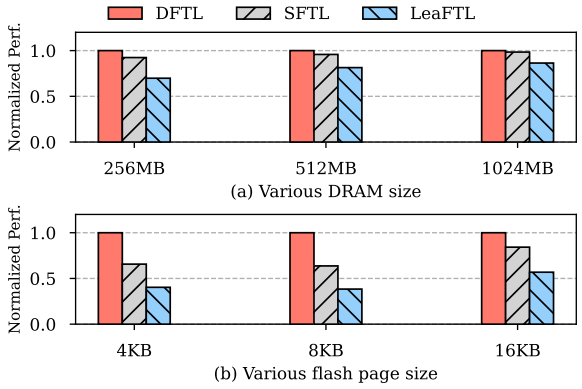


Figure 22: SSD performance with different DRAM capacity and flash page size (lower is better).

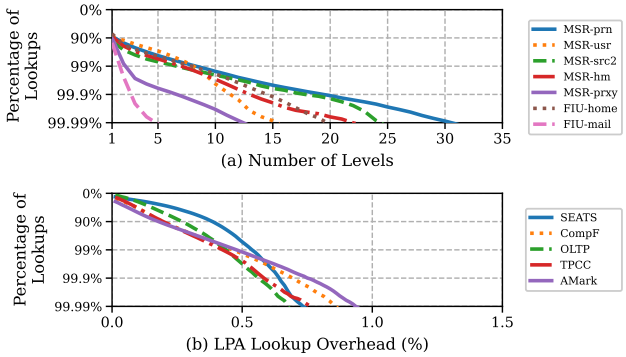


Figure 23: Performance overhead of the LPA lookup.

experiences a slight performance drop. As we fix the total SSD capacity and vary the page size, LeaFTL outperforms SFTL by 1.2× and 1.1× for the page size of 8KB and 16KB, respectively.

4.5 Overhead Source in LeaFTL

We evaluate the overhead sources in LeaFTL in three aspects: (1) the performance overhead of the learning procedure in LeaFTL; (2) the LPA lookup overhead in the learned segments; and (3) the overhead caused by the address misprediction in LeaFTL.

We evaluate the performance of segment learning and address lookup on an ARM Cortex-A72 core. This core is similar to the storage processor used in modern SSDs. The learning time for a batch of 256 mapping entries is 9.8–10.8 μs (see Table 3). As we learn one batch of index segments for every 256 flash writes, the learning overhead is only 0.02% of their flash write latency.

In LeaFTL, the LPA lookup is 40.2–67.5 ns, as the binary search of segments is fast and some segments can be cached in the processor cache. The lookup time is slightly higher as we increase γ, due to the additional CRB accesses. We also profile the cumulative distribution function (CDF) of the number of levels to lookup for each LPA lookup, and present the results in Figure 23 (a). For most of the tested workloads, 90% of the mapping table lookup can be fulfilled at the topmost level, and 99% of the lookups are within 10 levels.

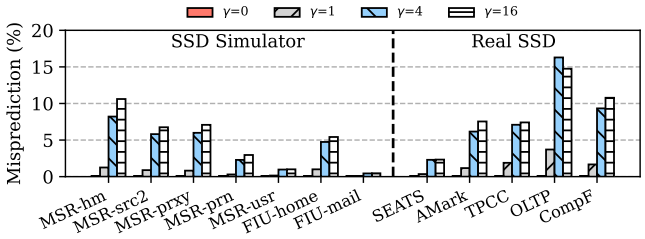


Figure 24: Misprediction ratio of flash pages access.

Table 3: Overhead source of LeaFTL with an ARM core.

γ	0	1	4
Learning (256 LPAs)	9.8 μs	10.8 μs	10.8 μs
Lookup (per LPA)	40.2 ns	60.5 ns	67.5 ns

Although MSR-prn workload requires more lookups than other workloads, it only checks 1.4 levels on average. We also evaluate the performance overhead of the LPA lookup on the real SSD, and show the results in Figure 23 (b). The extra lookup overhead for each flash read is 0.21% on average. And for 99.99% of all the lookups, the additional overhead is less than 1% of the flash access latency.

LeaFTL also has low misprediction ratios with approximate segments. This is because LeaFTL can still learn accurate segments even if $\gamma > 0$, and not all entries in the approximate segments will result in misprediction. As shown in Figure 24, most of the workloads achieve less than 10% misprediction ratio when $\gamma = 16$. We obtain similar misprediction ratio on the real SSD prototype. Note that each misprediction only incurs one flash read access with the help of our proposed OOB verification.

4.6 Impact on SSD Lifetime

The flash blocks of an SSD can only undergo a certain amount of writes. In this experiment, we use the write amplification factor (WAF, the ratio between the actual and requested flash writes) to evaluate the SSD lifetime. The SSD will age faster if the WAF is larger. As shown Figure 25, the WAF of LeaFTL is comparable to DFTL and SFTL. DFTL has larger WAF in most workloads. SFTL and LeaFTL occasionally flush translation pages to the flash blocks, but the cost is negligible.

5 DISCUSSION

Why Linear Regression. Unlike deep neural networks, the linear regression used in LeaFTL is simple and lightweight, which takes only a few microseconds to learn an index segment with embedded ARM processors available in modern SSD controllers. In addition, the linear regression algorithm has been well studied, and offers guaranteed error bounds for its learned results. LeaFTL is the first work that uses learning techniques to solve a critical system problem (i.e., address mapping) in SSDs.

Adaptivity of LeaFTL. LeaFTL focuses on the page-level address translation, its design and implementation will not be affected by the low-level flash memory organization (i.e., TLC/QLC). As we use TLC/QLC technique to further increase the SSD capacity, the address mapping issue will become more critical, since the SSD

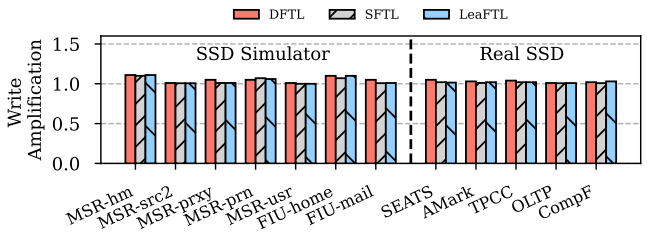


Figure 25: Write amplification factor of LeaFTL.

DRAM capacity does not scale well and becomes the bottleneck for caching address mappings and user data.

Recovery of Learned Index Segments. As discussed in §3.8, using a battery or large capacitor to preserve and persist the cached segments upon failures or crashes will simplify the recovery procedure significantly. In our real SSD prototype, we do not assume the battery-backed DRAM is available. Thus, we follow the conventional recovery approach in modern SSDs [20, 23], and scan flash blocks in parallel by utilizing the channel-level parallelism.

When we run real workloads like TPCC on the SSD prototype, we intentionally reboot the system after running the workload for a period of time (0.5-3 hours). We find that the system can recover in 15.8 minutes on average whenever the reboot happens. This is similar to the time of recovering the conventional page-level mapping table in DFTL [20]. This is mostly caused by scanning the blocks in a channel (70MB/s per channel in our SSD prototype), and the time for reconstructing recently learned segments is relatively low (101.3 milliseconds on average). We believe the recovery time is not much of a concern as the recovery does not happen frequently in reality. And the recovery can be accelerated as we increase the channel-level bandwidth. In addition, if an SSD can tolerate more data losses, we can still ensure the crash consistency by only loading the stored index segments from flash chips, which requires minimum recovery time.

6 RELATED WORK

Address Translation for SSDs. A variety of FTL optimizations have been proposed [8, 12, 20, 25, 28, 34, 49, 50]. These works exploited the data locality of flash accesses to improve the cache efficiency of the mapping table. However, most of them were developed with human-driven heuristics. An alternative approach is to integrate application semantics into the FTL, such as content-aware FTL [7]. However, they were application specific and required significant changes to the FTL. LeaFTL is a generic solution and does not require application semantics in its learning. Researchers proposed to integrate the FTL mapping table into the host [18, 23, 26, 66]. Typical examples include DFS [26], Nameless writes [66], FlashMap [23], and FlatFlash [4]. LeaFTL is orthogonal to them and can be applied to further reduce their memory footprint.

Machine Learning for Storage. Recent studies have been using learning techniques to build indexes such as B-trees, log-structured merge tree, hashmaps, and bloom filters [11, 14, 15, 32, 33, 42] for in-memory datasets, identify optimal cache replacement and prefetching policies [40, 53, 56, 57], facilitate efficient storage harvesting [52], and drive the development of software-defined storage [24]. LeaFTL applies learning techniques to optimize the address mapping. However, unlike existing optimizations [43, 63] such as learned page table for virtual memory that used deep neural networks to learn the patterns, LeaFTL provides a lightweight solution.

SSD Hardware Development. For the recent SSD innovations [3, 17, 19, 47] like Z-SSD [55], KVSSD [35], and ZNS SSD [21], DRAM capacity and storage processor are still the main constraints in SSD controllers. As we scale the storage capacity, the challenge with the address translation becomes only worse. Researchers recently deployed hardware accelerators inside SSD controllers for near-data computing [36, 41, 54, 58]. We wish to extend LeaFTL with in-storage accelerators to deploy more powerful learning models as the future work.

7 CONCLUSION

We present a learning-based flash translation layer, named LeaFTL for SSDs. LeaFTL can automatically learn different flash access patterns and build space-efficient indexes, which reduces the address mapping size and improves the caching efficiency in the SSD controller. Our evaluation shows that LeaFTL improves the SSD performance by 1.4× on average for a variety of storage workloads.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their helpful comments and feedback. This work is partially supported by the NSF CAREER Award 2144796, CCF-1919044, and CNS-1850317.

REFERENCES

[1] 2019. A Closer Look At SSD Power Loss Protection. https://www.kingston.com/en/blog/servers-and-data-centers/ssd-power-loss-protection.
[2] 2020. Harnessing Microcontrollers to Deliver Intelligent SSD Power Management and PLP Capabilities. https://www.atpinc.com/de/about/stories/microcontroller-SSD-power-loss-protection.
[3] 3D NAND – An Overview. 2022. https://www.simms.co.uk/tech-talk/3d-nand-overview/.
[4] Ahmed Abulila, Vikram Sharma Mailthoday, Zaid Qureshi, Jian Huang, Nam Sung Kim, Jin jun Xiong, and Wen mei Hwu. 2019. FlatFlash: Exploiting the Byte-Accessibility of SSDs within A Unified Memory-Storage Hierarchy. In *Proceedings of the 24th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'19)*. Providence, RI.
[5] Nitin Agrawal, Vijayan Prabhakaran, Ted Wobber, John D. Davis, Mark Manasse, and Rina Panigrahy. 2008. Design Tradeoffs for SSD Performance. In *Proceedings of the USENIX 2008 Annual Technical Conference (ATC'08)*. Boston, Massachusetts.
[6] Yu Cai, Saugata Ghose, Erich F Haratsch, Yixin Luo, and Onur Mutlu. 2017. Error characterization, mitigation, and recovery in flash-memory-based solid-state drives. *Proc. IEEE* 105, 9 (2017), 1666–1704.
[7] Feng Chen, Tian Luo, and Xiaodong Zhang. 2011. CAFTL: A Content-Aware Flash Translation Layer Enhancing the Lifespan of Flash Memory based Solid State Drives. In *Proceedings of the 9th USENIX Conference on File and Storage Technologies (FAST'11)*. San Jose, CA.
[8] Renhai Chen, Zhiwei Qin, Yi Wang, Duo Liu, Zili Shao, and Yong Guan. 2014. On-demand block-level address mapping in large-scale NAND flash storage systems. *IEEE Trans. Comput.* 64, 6 (2014), 1729–1741.
[9] Tae-Sun Chung, Dong-Joo Park, and Jongik Kim. 2011. LSTAFF*: An Efficient Flash Translation Layer for Large Block Flash Memory. In *Proceedings of the 2011 ACM Symposium on Applied Computing (SAC'11)*. TaiChung Taiwan.
[10] Curtis R Cook and Do Jin Kim. 1980. Best sorting algorithm for nearly sorted lists. *Commun. ACM* 23, 11 (1980), 620–624.
[11] Yifan Dai, Yien Xu, Aishwarya Ganesan, Ramnatthan Alagappan, Brian Kroth, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. 2020. From WiscKey to Bourbon: A Learned Index for Log-Structured Merge Trees. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI'20)*. Virtual Event.
[12] Niv Dayan, Philippe Bonnet, and Stratos Idreos. 2016. GeckoFTL: Scalable Flash Translation Techniques For Very Large Flash Devices. In *Proceedings of the International Conference on Management of Data (SIGMOD'16)*. San Francisco, CA.

[13] Djellel Eddine Difallah, Andrew Pavlo, Carlo Curino, and Philippe Cudré-Mauroux. 2013. OLTP-Bench: An Extensible Testbed for Benchmarking Relational Databases. *PVLDB* 7, 4 (2013).

[14] Paolo Ferragina, Fabrizio Lillo, and Giorgio Vinciguerra. 2020. Why Are Learned Indexes So Effective?. In *Proceedings of the 37th International Conference on Machine Learning (ICML'20)*. Virtual Event.

[15] Paolo Ferragina and Giorgio Vinciguerra. 2020. The PGM-Index: A Fully-Dynamic Compressed Learned Index with Provable Worst-Case Bounds. *Proceedings of the VLDB Endowment* 13, 8 (April 2020).

[16] FIU. 2009. FIU Server Traces.

[17] Flash Memory. 2022. https://en.wikipedia.org/wiki/Flash_memory.

[18] Fusion-io Directcache: Transparent Storage Accelerator. 2011. http://www.fusionio.com/systems/directcache/.

[19] Gartner. 2017. Forecast Overview: NAND Flash, Worldwide, 2017. https://www.gartner.com/doc/3745121/forecast-overview-nand-flash-worldwide

[20] Aayush Gupta, Youngjae Kim, and Bhuvan Urgaonkar. 2009. DFTL: A Flash Translation Layer Employing Demand-based Selective Caching of Page-level Address Mappings. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'09)*. Washington, DC.

[21] Kyuhwa Han, Hyunho Gwak, Dongkun Shin, and Joo-Young Hwang. 2021. ZNS+: Advanced Zoned Namespace Interface for Supporting In-Storage Zone Compaction. In *15th {USENIX} Symposium on Operating Systems Design and Implementation (OSDI'21)*. 147–162.

[22] Jian Huang, Anirudh Badam, Laura Caulfield, Suman Nath, Sudipta Sengupta, Bikash Sharma, and Moinuddin K. Qureshi. 2017. FlashBlox: Achieving Both Performance Isolation and Uniform Lifetime for Virtualized SSDs. In *Proceedings of the 15th Usenix Conference on File and Storage Technologies (FAST'17)*. Santa clara, CA.

[23] Jian Huang, Anirudh Badam, Moinuddin K. Qureshi, and Karsten Schwan. 2015. Unified Address Translation for Memory-mapped SSDs with FlashMap. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA'15)*. Portland, OR.

[24] Jian Huang, Daixuan Li, and Jinghan Sun. 2022. Learning to Drive Software-Defined Storage. *Workshop on Machine Learning for Systems at NIPS'22* (2022).

[25] Song Jiang, Lei Zhang, XinHao Yuan, Hao Hu, and Yu Chen. 2011. S-FTL: An Efficient Address Translation for Flash Memory by Exploiting Spatial Locality. In *Proceedings of the 2011 IEEE 27th Symposium on Mass Storage Systems and Technologies (MSST'11)*. IEEE Computer Society.

[26] William K. Josephson, Lars A. Bongo, Kai Li, and David Flynn. 2010. DFS: A File System for Virtualized Flash Storage. *ACM Trans. on Storage* 6, 3 (2010), 14:1–14:25.

[27] Jun He, Sudarsun Kannan, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau. 2017. The Unwritten Contract of Solid State Drives. In *Proceedings of the Twelfth European Conference on Computer Systems (EuroSys'17)*. Belgrade, Serbia.

[28] Dawoon Jung, Jeong-UK Kang, Heeseung Jo, Jin-Soo Kim, and Joonwon Lee. 2010. Superblock FTL: A superblock-based flash translation layer with a hybrid address translation scheme. *ACM Transactions on Embedded Computing Systems (TECS)* 9, 4 (2010), 1–41.

[29] Jeong-Uk Kang, Heeseung Jo, Jinsoo Kim, and Joonwon Lee. 2006. A Superblock-Based Flash Translation Layer for NAND Flash Memory. In *Proceedings of the 6th International Conference on Embedded Software (EMSOFT'06)*. Seoul, South Korea.

[30] Luyi Kang, Yuqi Xie, Weiwei Jia, Xiaohao Wang, Jongryool Kim, Changhwan Youn, Myeong Joon Kang, Jin Lim, Bruce Jacob, and Jian Huang. 2021. IceClave: A Trusted Execution Environment for In-Storage Computing. In *Proceedings of the 54th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'21)*. Virtual Event.

[31] Jesung Kim, Jong Min Kim, S.H. Noh, Sang Lyul Min, and Yookun Cho. 2002. A space-efficient flash translation layer for CompactFlash systems. *IEEE Transactions on Consumer Electronics* 48, 2 (2002).

[32] Andreas Kipf, Ryan Marcus, Alexander van Renen, Mihail Stoian, Alfons Kemper, Tim Kraska, and Thomas Neumann. 2020. RadixSpline: A Single-Pass Learned Index. In *Proceedings of the Third International Workshop on Exploiting Artificial Intelligence Techniques for Data Management (aiDM '20)*. Portland, Oregon.

[33] Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, and Neoklis Polyzotis. 2018. The Case for Learned Index Structures. In *Proceedings of the 2018 International Conference on Management of Data (SIGMOD'18)*. Houston, TX, USA.

[34] Hunki Kwon, Eunsam Kim, Jongmoo Choi, Donghee Lee, and Sam H Noh. 2010. Janus-FTL: Finding the optimal point on the spectrum between page and block mapping schemes. In *Proceedings of the tenth ACM international conference on Embedded software*. 169–178.

[35] Samsung Memory Solutions Lab. 2017. Samsung Key Value SSD enables High Performance Scaling. https://www.samsung.com/semiconductor/global.semi.static/Samsung_Key_Value_SSD_enables_High_Performance_Scaling-0.pdf (2017).

[36] Joo Hwan Lee, Hui Zhang, Veronica Lagrange, Praveen Krishnamoorthy, Xiaodong Zhao, and Yang Seok Ki. 2020. SmartSSD: FPGA accelerated near-storage data analytics on SSD. *IEEE Computer architecture letters* 19, 2 (2020), 110–113.

[37] Sungjin Lee, Ming Liu, Sangwoo Jun, Shuotao Xu, Jihong Kim, and Arvind. 2016. Application-managed flash. In *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST'16)*. 339–353.

[38] Sungjin Lee, Dongkun Shin, Young-Jin Kim, and Jihong Kim. 2008. LAST: Locality-Aware Sector Translation for NAND Flash Memory-Based Storage Systems. *In Proceedings of the SIGOPS Operating Systems Review* (2008).

[39] Sang-Won Lee, Dong-Joo Park, Tae-Sun Chung, Dong-Ho Lee, Sangwon Park, and Ha-Joo Song. 2007. A Log Buffer-Based Flash Translation Layer Using Fully-Associative Sector Translation. *ACM Transactions on Embedded Computing Systems* 6, 3 (2007), 18:1–18:27.

[40] Evan Liu, Milad Hashemi, Kevin Swersky, Parthasarathy Ranganathan, and Junwhan Ahn. 2020. An imitation learning approach for cache replacement. In *International Conference on Machine Learning*. PMLR, 6237–6247.

[41] Vikram Sharma Mailthoday, Zaid Qureshi, Weixin Liang, Ziyan Feng, Simon Garcia de Gonzalo, Youjie Li, Hubertus Franke, Jinjun Xiong, Jian Huang, and Wen mei Hwu. 2019. DeepStore: In-Storage Acceleration for Intelligent Queries. In *Proceedings of the 52nd IEEE/ACM International Symposium on Microarchitecture (MICRO'19)*. Columbus, OH.

[42] Ryan Marcus, Emily Zhang, and Tim Kraska. 2020. CDFShop: Exploring and Optimizing Learned Index Structures. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD'20)*. Portland, OR, USA. https://doi.org/10.1145/3318464.3384706

[43] Artemiy Margaritov, Dmitri Ustiugov, Edouard Bugnion, and Boris Grot. 2018. Virtual Address Translation via Learned Page Table Indexes. In *Proceedings of the Workshop on ML for Systems at NeurIPS*. Montreal, Canada.

[44] Kiran Kumar Matam, Gunjae Koo, Haipeng Zha, Hung-Wei Tseng, and Murali Annavaram. 2019. GraphSSD: Graph Semantics Aware SSD. In *Proceedings of the 46th International Symposium on Computer Architecture (ISCA'19)*. Phoenix, Arizona.

[45] Microsoft. 2007. MSR Cambridge Traces.

[46] Jian Ouyang, Shiding Lin, Song Jiang, Yong Wang, Wei Qi, Jason Cong, and Yuanzheng Wang. 2014. SDF: Software-Defined Flash for Web-Scale Internet Storage Systems. In *Proceedings of 19th International Conference on Architectural Support for Programming Language and Operating Systems (ASPLOS'14)*. Salt Lake City, UT.

[47] Over 50 years of development history of Flash Memory Technology. 2019. https://www.elinfor.com/knowledge/over-50-years-of-development-history-of-flash-memory-technology-p-11271.

[48] Nikolaos Papandreou, Haralampos Pozidis, Nikolas Ioannou, Thomas Parnell, Roman Pletka, Milos Stanisavljevic, Radu Stoica, Sasa Tomic, Patrick Breen, Gary Tressler, et al. 2020. Open block characterization and read voltage calibration of 3D QLC NAND flash. In *2020 IEEE International Reliability Physics Symposium (IRPS)*. IEEE, 1–6.

[49] Chanik Park, Wonmoon Cheon, Jeonguk Kang, Kangho Roh, Wonhee Cho, and Jin-Soo Kim. 2008. A reconfigurable FTL (flash translation layer) architecture for NAND flash-based applications. *ACM Transactions on Embedded Computing Systems (TECS)* 7, 4 (2008), 1–23.

[50] Zhiwei Qin, Yi Wang, Duo Liu, and Zili Shao. 2010. Demand-based block-level address mapping in large-scale NAND flash storage systems. In *Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*.

[51] Benjamin Reidys, Peng Liu, and Jian Huang. 2022. RSSD: Defend against Ransomware with Hardware-Isolated Network-Storage Codesign and Post-Attack Analysis. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'22)*. Lausanne, Switzerland.

[52] Benjamin Reidys, Jinghan Sun, Anirudh Badam, Shadi Noghabi, and Jian Huang. 2022. BlockFlex: Enabling Storage Harvesting with Software-Defined Flash in Modern Cloud Platforms. In *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI'22)*. Carlsbad, CA.

[53] Liana V Rodriguez, Farzana Yusuf, Steven Lyons, Eysler Paz, Raju Rangaswami, Jason Liu, Ming Zhao, and Giri Narasimhan. 2021. Learning Cache Replacement with CACHEUS. In *19th USENIX Conference on File and Storage Technologies (FAST'21)*. 341–354.

[54] Zhenyuan Ruan, Tong He, and Jason Cong. 2019. INSIDER: Designing In-Storage Computing System for Emerging High-Performance Drive. In *Proceedings of the 2019 USENIX Annual Technical Conference (USENIX ATC'19)*. Renton, WA.

[55] Samsung Z-NAND. 2019. https://www.samsung.com/semiconductor/ssd/z-ssd/.

[56] Subhash Sethumurugan, Jieming Yin, and John Sartori. 2021. Designing a Cost-Effective Cache Replacement Policy using Machine Learning. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 291–303.

[57] Zhan Shi, Xiangru Huang, Akanksha Jain, and Calvin Lin. 2019. Applying deep learning to the cache replacement problem. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. 413–425.

[58] smartssd 2018. SmartSSD Computational Storage Drive. https://www.xilinx.com/applications/data-center/computational-storage/smartssd.html.

[59] Vasily Tarasov, Erez Zadok, and Spencer Shepler. 2016. Filebench: A flexible framework for file system benchmarking. *The USENIX Magazine* 41, 1 (2016).

[60] Usman Saleem, Advanced SSD Buying Guide - NAND Types, DRAM Cache, HMB Explained. 2022. https://appuals.com/ssd-buying-guide/.

[61] Dana Van Aken, Djellel E. Difallah, Andrew Pavlo, Carlo Curino, and Philippe Cudré-Mauroux. 2015. BenchPress: Dynamic Workload Control in the OLTP-Bench Testbed. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD'15)*.

[62] Xiaohao Wang, Yifan Yuan, You Zhou, Chance C. Coats, and Jian Huang. 2019. Project Almanac: A Time-Traveling Solid-State Drive. In *Proceedings of the 14th European Conference on Computer Systems (EuroSys'19)*. Dresden, Germany.

[63] Nan Wu and Yuan Xie. 2021. A Survey of Machine Learning for Computer Architecture and Systems. *CoRR* abs/2102.07952 (2021). https://arxiv.org/abs/2102.07952

[64] Qing Xie, Chaoyi Pang, Xiaofang Zhou, Xiangliang Zhang, and Ke Deng. 2014. Maximum Error-Bounded Piecewise Linear Representation for Online Stream Approximation. *Proceedings of the VLDB Journal* 23, 6 (Dec. 2014).

[65] Qing Xie, Chaoyi Pang, Xiaofang Zhou, Xiangliang Zhang, and Ke Deng. 2014. Maximum error-bounded piecewise linear representation for online stream approximation. *The VLDB journal* 23, 6 (2014), 915–937.

[66] Yiying Zhang, Leo Prasath Arulraj, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2012. De-indirection for Flash-based SSDs with Nameless Writes. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies (FAST'12)*. San Jose, CA.