A Practical Learning-Based FTL for Memory Constrained Mobile Flash Storage

Zelin Du[†], Kecheng Huang[†], Tianyu Wang[‡], Xin Yao[§], Renhai Chen [*§], and Zili Shao[†]
[†]Department of Computer Science and Engineering, The Chinese University of Hong Kong, Hong Kong SAR
[‡]College of Computer Science and Software Engineering, Shenzhen University, Shenzhen, China
[§]Huawei Technologies, China
zldu22, kchuang21, shao@cse.cuhk.edu.hk, tywang@szu.edu.cn, yao.xin1, chenrenhai@huawei.com

***Abstract*—The rapidly growing mobile market is pushing flash storage manufacturers to expand capacity into the terabyte range. However, this presents a significant challenge for mobile storage management: more logical-to-physical page mappings are desired to be efficiently managed and cached while the available caching space is extremely limited. This motivates us to shift toward a new learning-based paradigm: rather than maintaining mappings for individual pages, the learning-based approach can represent mapping relationships for a set of continuous pages. However, to construct linear models, existing methods that either consume the already-limited memory space or reuse flash garbage collection demonstrate poor model construction capabilities or significantly degrade flash performance, making them impractical for real-world use. In this paper, we propose LFTL, a practical, learning-based on-demand flash translation layer design for flash management in mobile devices. In contrast to prior work that centered around gathering sufficient mappings for linear model construction, our key insight is that linear patterns can be extracted and refined by leveraging the orderly, LPA-aligned write stream typical of mobile devices. By doing this, highly accurate linear models can be constructed regardless of the constraints of mobile device's cache limitation. We have implemented a fully functional prototype of LFTL based on FEMU. Our evaluation results show that LFTL is more adaptable to memory-constrained storage devices than state-of-the-art learning-based approaches.**

***Index Terms*—Mobile flash storage, flash translation layer, learned index**

I. Introduction

In recent years, flash memory has become a dominant solution for providing storage capability in mobile devices, including smartphones, tablets, and wearables [1]–[3]. A critical component in managing flash memory is the page-level On-demand Flash Translation Layer (DFTL), which handles the logical-to-physical mapping of page addresses and on-demandingly preserves mapping entries in memory through a Cached Mapping Table (CMT) [4]–[8], enabling faster access for upper-layer applications.

In today's booming mobile market, the rising demand for device storage space is driving manufacturers to expand capacity to terabytes while the growth of devices' cache capacity has not kept pace (e.g., 2MiB of SRAM) [9]–[11]. This poses an unprecedented challenge for mobile device DFTL design: the ever-increasing storage space necessitates more page addresses to be managed by DFTL while the benefits of on-demand caching are severely constrained by the limited cache space. For example, for a mobile device with 1TiB storage capacity, only 0.1% of mapping entries (at 4KiB granularity) can be cached in a 2MiB CMT. Each CMT miss requires retrieving a 4KiB page containing the needed mapping entry, which not only causes I/O amplification but also increases application response latency.

In this paper, we shift towards a more promising paradigm: the learning-based DFTL. Rather than caching individual page mapping entries in the CMT, a linear model that covers a range of mappings demonstrates a superior ability to preserve more mappings while occupying less space in the CMT. However, making the learning-based DFTL a practical solution is challenging, especially considering the real-world requirements of mobile devices. We outline three key issues below.

Issue 1: Lack of Observations in Mobile Devices. The linear model can abstract relationships between sorted keys and values. For instance, one can construct a linear model, such as PPA = f(LPA), to represent the linear correlation between Logical Page Addresses (LPAs, e.g., 0x1, 0x2) and Physical Page Addresses (PPAs, e.g., 0x8, 0x9). To expand the linear model's expression range and minimize deviations, it is essential to accumulate and observe a sufficient number of sorted LPAs. However, this is nearly impossible in mobile device environments, where available memory for such accumulation is extremely limited. Specifically, LeaFTL [12] proposes generating linear models based solely on the observation window provided by the SSD's write buffer. Due to the tight memory budget of mobile flash devices, this approach restricts the model's ability to cover more mappings and is challenging to apply in practical DFTL design. LearnedFTL [13] suggests leveraging garbage collection as the observation window for linear index construction. However, this contradicts with design purpose of garbage collection where minimal numbers of valid pages are supposed to be reclaimed to reduce write amplification while the linear model desires sufficient valid pages for better construction.

Issue 2: Disturbance of Outliers. Considering the observed mapping entries are (already) insufficient to tolerate deviations, the disturbance caused by outliers is becoming more severe: the real-world workloads (operations to LPAs) are not "perfectly sorted" where outliers may unexpectedly pollute an entire linear model. Such outliers are operations to random LPAs, which may cut in the line of the write stream (i.e., sorted LPAs) and impact the accuracy of the linear regression of linear models.

* Renhai Chen is the corresponding author.

978-3-9826741-0-0/DATE25/© 2025 EDAA

How to mitigate the disturbance of outliers in an already limited observation window is another critical challenge.

Issue 3: Competition among Different Sources. The cache of flash memory, which is desired to be utilized to observe sorted LPAs for constructing linear models (i.e., temporarily holds mapping entries in cache), is also responsible for (1) on-demandingly preserving linear model parameters, (2) mapping entries of outliers, and (3) always keeping many other critical metadata (global translation directory, block control metadata, etc) for flash management. This makes the already limited memory budget even more exhausting.

In this paper, we propose LFTL, a practical, learning-based DFTL design for flash management in mobile devices to address the aforementioned critical issues. Essentially, the fundamental gap between learning-based approaches and mobile device storage FTL design is that the former requires substantial memory space to observe correlation patterns, whereas modern mobile devices offer only a limited memory budget despite having a large number of pages to map.

To bridge this gap, instead of using costly cache space for linear pattern observation, a critical insight is that, *the linear pattern can be extracted and refined by leveraging the orderly, LPA-aligned write stream typical of mobile devices*. Specifically, the log-structured file systems adopted by mobile devices (represented by F2FS [14]) naturally divide write requests into multiple distinct write streams. Inside each stream, write requests are continuously assigned to orderly, LPA-aligned spaces. This brings us a valuable opportunity to leverage such ready-made sorted patterns to generate linear models without occupying cache space for observation. And the saved space can preserve more linear models for on-demand caching. This opens the door for the learning-based DFTL design space to escape from the cache space restriction and embrace a more promising direction: by saving cache space previously used for model observation, more deviations can be captured, allowing to fine-tune the linear model for expanded expression range.

To achieve this, we propose three techniques. First, we propose a spatial partitioning scheme to separate concurrent write streams into subspaces for generating monotonically increasing LPAs. To form consecutive PPAs, we propose a linear group concept where each linear group is a collection of consecutive flash blocks. Based on the well-prepared sorted LPAs and PPAs, we propose a set of refining approaches to strengthen the fault tolerance capability of the linear regression algorithm. Inspired by the RANSAC (RANdom SAmple Consensus) [15], we collect the hints (i.e. outlier ratio) and exemplary sequence per linear group to facilitate the convergence. The point mapping entries of the outliers should be stored together with the linear model, and they will be loaded on demand. We propose a chunk-based outlier management scheme for efficient outlier storage, which can mitigate the potential degradation when facing random LPA patterns.

We have implemented a full-functional prototype of LFTL based on FEMU [16], and compare LFTL with the state-of-the-art learning-based FTL approaches, including DFTL, LeaFTL, and LearnedFTL. The evaluation results show that LFTL shows preferable adaptability to memory-constrained storage devices where LFTL outperforms the state-of-the-art approaches by up to 1.3× in terms of throughput.

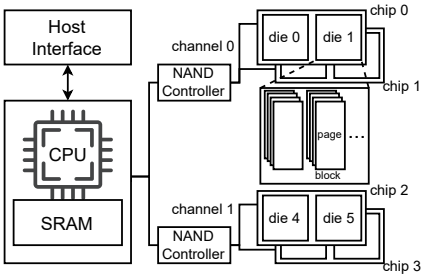


Fig. 1. Internals of the UFS device.

TABLE I
SSD VS. UFS

	SSD	UFS
Typical volume	1TiB	256GiB
Bandwidth	10GiB/s@PCIE5.0	4GiB/s@UFS4.0
Typical memory size	1GiB@DRAM	2MiB@SRAM
L2P map caching	fully loaded/on-demand paging	on-demand paging

II. Background and Motivation

A. Memory-Constrained Mobile Flash Storage

As depicted in Figure 1, the mobile flash storage device comprises a main controller, limited volatile memory, and flash memories. UFS [17] serves as a standard interface for contemporary mobile flash storage. The flash translation layer, operated by the controller, facilitates the conversion of logical page addresses (LPA) to physical page addresses (PPA). By fully populating the logical-to-physical (L2P) mapping table in memory, the FTL can swiftly execute the translation process. However, this performance-enhancing approach is unfeasible for UFS. As illustrated in Table I, UFS lacks DRAM and possesses only minimal SRAM. In a typical UFS device, a mere 0.3% of the mapping table can be accommodated in memory.

B. Learning-based FTL

The studies on the learned index have demonstrated its potential for compressing the mapping table [18]–[21]. The requirement for using the learned index is to store the data in the key order. A linear model can represent the linear correlation between the keys and the positions. Benefiting from the small number of parameters of the linear model, the mapping table size can be greatly reduced. Sometimes, the accuracy requirements of the linear model are more relaxed, requiring only that the distance between the predicted position and the real position be within the error margin. For such an approximate linear model, the additional search must be done in the last mile around the predicted position. There are two state-of-the-art learning-based FTLs, LeaFTL [12] and LearnedFTL [13]. Unfortunately, none of them can be applied to the memory-constrained mobile flash storage.

LeaFTL aggressively replaces the mapping table with linear models, including accurate and approximate models. LeaFTL sorts the flash pages in ascending order based on their LPAs in the data buffer. When pages in the data buffer are flushed to the

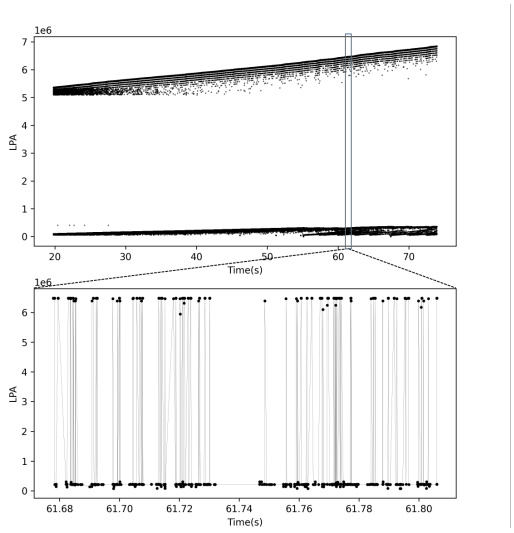
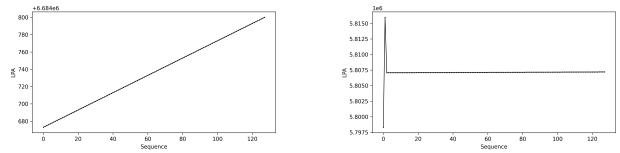


Fig. 2. LPA trace of filesever-filebench.



(a) perfect linear stream. (b) linear stream polluted by outliers.

Fig. 3. Linear streams in subspace.

flash memory, their PPAs are assigned in ascending order. The desired data buffer to accumulate sufficient sequential LPAs is 8MiB (2kilo 4KiB pages), which is unaffordable on mobile flash devices. If the data buffer size is very limited, there is no opportunity to observe a long linear LPA sequence.

LearnedFTL is proposed to avoid the last-mile search overhead caused by approximate linear models. It mainly relies on garbage collection (GC) to write valid pages using continuous flash memory to build accurate linear models. LearnedFTL equips the linear models with a bitmap filter, indicating whether a certain prediction for an LPA is accurate. The L2P mapping entries are reserved to handle the inaccurate prediction via an on-demand cache. LearnedFTL uses model-embedded GTD to pin the linear models and bitmap filters in memory. Each GTD entry has an additional 128B model and bitmap. For instance, 8MiB memory is required to store model-embedded GTD for a 128GiB device. Keeping the model-embedded GTD in small on-device memory is impossible.

C. Locality of Write Requests

Previous studies have explored various methods for establishing linear correlations between LPAs and PPAs. However, these methods have been either too proactive or too reactive in handling foreground write requests. Our observation indicates that the spatial and temporal locality inherent in the sequence of foreground write requests can be leveraged organically to establish linear correlations in their L2P mappings.

We use the blktrace tools [22] to capture the block access trace of the fileserver-filebench [23] on the mobile platform (F2FS on Linux kernel 5.4). As shown in Figure 2, we plot the LPAs of write requests in a time series. Multiple linear address

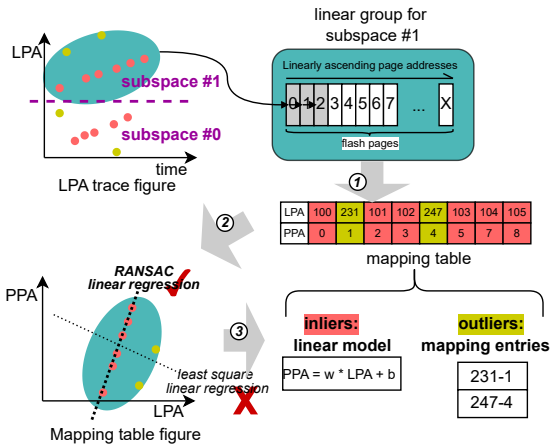


Fig. 4. Workflow of LFTL.

sequences being written simultaneously, referred to as write streams, are present. The host interleaves write requests among these write streams within a short time window. Due to the presence of multiple hotspots interspersed with each other, the storage device perceives spatial locality as fragmented. Despite the assigned PPAs being sequential, the controller processes the write requests individually, disrupting the potentially sustainable linear correlations of the L2P mappings.

The key of obtaining a sustainable linear LPA sequence is to isolate each write stream from the multi-stream request sequence. Space partitioning that leaves only one data stream on each subspace can isolate write streams. Within each subspace, physical addresses are assigned sequentially. When the linear LPA sequence aligns with consecutive LPAs within a subspace, the L2P mappings can be represented using a linear model.

However, obtaining the desired precise linear models is challenging, as the subspace not only consists of the write stream but also includes random writes. Two sampled LPA traces from different 256MB-sized subspaces are shown in Figure 3, the trace in Figure 3(a) is a perfect linear stream, the head of the trace in Figure 3(b) is polluted by random writes. The accuracy of the linear model is influenced by the distribution of the training data. To improve accuracy, it is essential to filter out the random writes from the training data.

III. LFTL: Overview

We propose the LFTL to achieve the goal that using the linear model reduces the L2P mapping table size on memory-constrained mobile flash storage. Since the out-of-place update manner of the flash memory, the write requests create new L2P mappings. We utilize the write requests to reshape the L2P mappings to be linearly correlated. Our idea is shown in Figure 4. The write streams are extracted by the space partitioning and logged in the linear groups (Section IV-A). Then the mappings are fed into a special linear regression algorithm to generate the accurate linear model and filter out the random writes. The LPA of the outliers are recorded by the outlier identifier and their L2P mappings are recorded by the point mapping entries (Section IV-B). Finally, the linear models, outlier identifiers, and point mapping entries will be loaded in memory on-demand and will be managed with different priorities (Section IV-C).

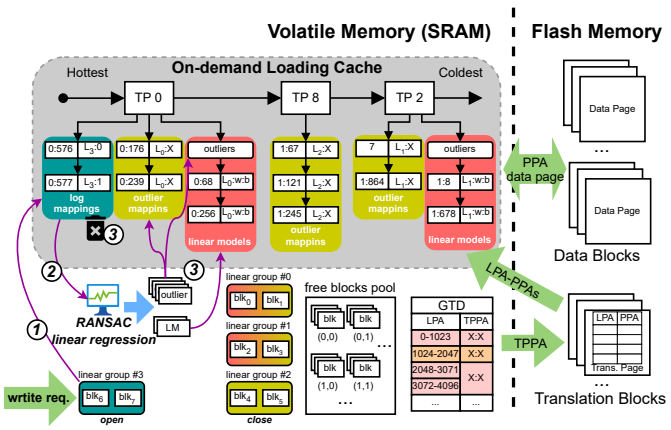


Fig. 5. Architectural overview of LFTL.

IV. Techniques

A. Space Partition and Linear Group

Based on our observation (Section II-C), we partition the logical space into multiple disjoint subspaces to extract the write streams from the write requests to get the linear LPAs.

The default subspace size is 256MiB (64kilo 4KiB pages). From the perspective of the subspace, write requests arriving in time order are naturally sorted by LPA except for a few random writes. A set of flash blocks are organized together as a linear group. We use the PPAs of the blocks to represent a linear group: $LG = \{PPA_{B_0}, PPA_{B_1}, ...PPA_{B_{K-1}}\}$. When the write requests come, the corresponding subspace has an open linear group. To assign the linear PPAs, the pages of each write stream are logged in the linear group in an append-only manner. If the linear group doesn't have the empty flash page, the linear group will be marked as closed.

In the read scenarios, the bad blocks are continually emerging as the P/E cycle increases. The dynamic block allocation skips the bad block and collects the blocks from all parallel units (e.g. channels, chips, dies) evenly. The address sent to the flash controller to access flash memory is called the global PPA, and the address assigned to the request within the linear group is called the local PPA. The local PPAs are continuous to provide stable linear addresses. We can translate a local PPA to a global PPA by contacting it with the block PPA:

$$PPA_{global} = PPA_{local} + LG[PPA_{local}\%K] \quad (1)$$

The LPA to local PPA mappings in the open linear group are temporarily held in memory until the linear model is created (Section IV-C). At the appropriate time, e.g. the linear group is closed, the LPA to local PPA mappings should be fed into the linear regression algorithm as the training dataset to create the linear model (Section IV-B).

B. Robust Linear Regression

The normal linear regression, e.g. least squares, tries to find the best solution closest to all data points even though some data points are outliers of the linear distribution. We use a random sample consensus (RANSAC) linear regression algorithm to build linear models while filtering the outliers.

Algorithm 1 Pesude code of RANSAC algorithm

Require: Data points $\mathcal{P}$, Max iterations I, sample size K.
Ensure: Model M, Outliers $\mathcal{O}$

while $i \leq I$ **do**
  $S \leftarrow$ sample k points from $\mathcal{P}$
  $M \leftarrow$ Linear-regression(S)
  $\mathcal{O} \leftarrow$ check the M on $\mathcal{P}$ to get outliers
  if $|\mathcal{O}| < |\hat{\mathcal{O}}|$ **then**
    $\hat{M} \leftarrow M$
    $\hat{\mathcal{O}} \leftarrow \mathcal{O}$
  end if
  $i \leftarrow i + 1$
end while
$\hat{M} \leftarrow$ Linear-regression($P - \hat{\mathcal{O}}$)
$\hat{\mathcal{O}} \leftarrow$ check the $\hat{M}$ on $\mathcal{P}$ to get outliers using tight error bound (± 0.5)

Algorithm 2 Generate linear models

Require: Data points $\mathcal{P}$
Ensure: Models $\mathcal{M}$, Outliers $\mathcal{O}$

$M, \mathcal{O} \leftarrow$ RANSAC($\mathcal{P}$, 2, 2)
while $i < |\mathcal{P}|$ **do**
  if $Ppoint$ is None and $\mathcal{P}[i+1].x - \mathcal{P}[i].x \neq 1$ **then**
    $Ppoint \leftarrow \mathcal{P}[i]$
  else if $\mathcal{P}[i].x - Point.x \neq 1$ **then**
    if $\mathcal{P}[i]$ not in $\mathcal{O}$ **then**
      append $\{M, Spoint.x\}$ to $\mathcal{M}$
      $Spoint \leftarrow \mathcal{P}[i]$
      $Ppoint \leftarrow$ None
    end if
  else
    $Ppoint \leftarrow$ None
  end if
  $i \leftarrow i + 1$
end while

As shown in Algorithm 1, the RANSAC algorithm is an iterative method to estimate parameters of a mathematical model from a set of observed data that contains outliers, when outliers are to be accorded no influence on the values of the estimates. RANSAC assumes that, given a small set of sampled inliers, the model that fits them will fit all inliers. RANSAC uses the voting scheme to find the optimal result. The round of sampling, estimating, and voting is determined by the outlier rate of the dataset (p) and the desired probability of success (q) which is the probability of excluding all outliers. The iteration number to converge to an optimal inliers consensus set can be calculated by the following equation:

$$I = \lceil \frac{\ln(1-q)}{\ln(1-p^k)} \rceil \quad (2)$$

Early termination. Typically, q is set to 0.99. According to our observation, p is set to 0.95. When the sample size is 3, I is 3. We add an early termination scheme to reduce the number of iterations. The inliers are likely in the continuous regions. When the write request comes to the open linear group, if the LPA of the write request and its predecessor are consecutive, we add its LPA to local PPA mapping to the exemplary set. We add new members to the exemplary set until the number is equal to k (the sample size in RANSAC). In the first iteration of RANSAC, we will use the exemplary set instead of the random sample set. If it works well, then terminate the iteration.

Refine model. After converging to the inliers consensus set, we improve the model by re-estimating it on the consensus set. Then, we use a stricter error bound to identify all data points

with errors larger than ± 0.5 as outliers. So that, the rounding-up result of the model is precise for all inliers.

Segment model. For ease of management, we require that the LPAs covered by each linear model must be continuous. For a given LPA, it can only be in the range of one linear model. The procedure of segmenting the model is shown in Algorithm 2. After getting the initial accurate linear model by the RANSAC algorithm, we segment the LPA range of the linear model at the LPA discontinuity point to create sub-models for each continuous LPA range. The linear function ($(y = w * x + b)$) of the initial model is inherited by the sub-models.

C. Mapping Table Management

In addition to the parameters of the linear function (w, b), each linear model also contains the necessary metadata, i.e. start LPA, and group ID. The parameters w and b are 2B-float variables. The logical space is partitioned into subspaces, the length of LPA inside the subspace is 2B. The group ID is also a 2B short integer. The linear model is the same size as an 8-byte point mapping entry. The mapping table holds both linear models and point mapping entries. The whole mapping table is stored in translation pages and should be loaded in memory on demand. The linear model is used in conjunction with the outlier identifier which can be a bitmap or a LPA array. The bitmap labels the outliers by setting the corresponding position to 1. The LPA array stores the LPA of outliers. Both formats are used as needed, we use bitmap when there are a lot of outliers and vice versa with the LPA array.

A chunk is a group of continuous flash pages whose L2P mappings are recorded on the same translation page. Since the linear models reduce the records stored in the mapping table, the necessary translation pages to store the records are supposed to be reduced. Originally, each GTD entry corresponds to a translation page. If the translation pages are reduced, in some LPA ranges, one translation page can hold the records for many GTD entries. For example, in Figure 5 two GTD neighbor entries 2045 3071 and 3072 4096 shared the same translation page. The translation page can be split or merged with its GTD neighbors according to the number of entries to be stored.

Chunk-level outlier management. As shown in Figure 5, like TPFTL [24], we use a two-level LRU list to chain the translation page nodes (TP node) with the mapping entries in memory. Each TP node has up to 3 secondary lists, the outlier mappings list, the linear models list, and the log mappings list for holding the untrained mappings of the open linear group. For a given LPA, we first get its translation page number to find the TP node. Then get its PPA from the log mappings list, if exists. Otherwise, we check the outlier identifier at the head of the linear model list to decide which list it might be. Finally, the PPA will be obtained from the corresponding list, either in the linear models or in the outlier mappings. If the target entry is not in memory, the TP node and lists will be loaded on demand with a perfecting policy. The first eviction candidate is the first clean entry from the tail of the coldest outlier mappings list. If there is no clean entry, all the dirty entries of the coldest TP node will be flushed to the translation page and marked as clean. If the outlier mappings list is empty, the linear model at the tail of the list should be evicted. The outlier identifier will be evicted following the last linear model.

TABLE II
FLASH DEVICE CONFIGURATION.

Parameters	Value	Parameters	Value
Page read	40μ s	# of channels	8
Page write (program)	200μ s	# of chips per channel	8
Block erase	2ms	Capacity	32GiB
Page size	4KiB	SRAM size	2MiB
Block size	2MiB	bad block ratio	1%

TABLE III
MEMORY ALLOCATION.

SRAM assignment	LeaFTL	LearnedFTL
No-reserve	1.5MiB model cache 0.5MiB data buffer	1.5MiB GTD 0.5MiB CMT
Half-reserve	0.75MiB model cache 0.25MiB data buffer	\

V. EVALUATION

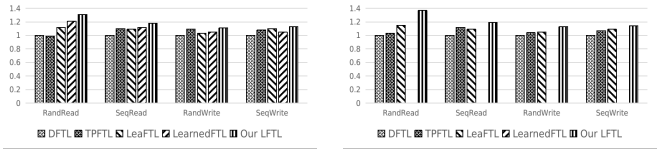
A. Experimental Setup

The experiments are conducted on FEMU [16], a QEMU-based and DRAM-backed SSD emulator that is widely used in SSD studies [13], [25], [26]. It runs on a machine with Intel i9-10850K 3.60GHz CPU (10-core/20-thread) and 64GB DRAM. The kernel is Linux 5.4 and the filesystem is F2FS, which is identical to the system environment of mobile devices (i.e. Android). The SSD settings are configured to align with the UFS. The UFS has a similar multi-channel and multi-way architecture to SSD, however, the on-device memory size is much smaller. The configuration of the flash storage device is shown in Table II. To mimic the actual scenario, we randomly marked 1% of the blocks as bad blocks and supplemented the capacity with provisioning blocks. Normally, the SRAM in the UFS controller is 2MiB [9], half (1MiB) is the mapping cache, and the other half is reserved for the firmware.

LFTL is compared against two types of page-level FTLs, the classic on-demand loading FTLs (DFTL [4], TPFTL [24]) and the learning-based FTLs (LeaFTL [12] and LearnedFTL [13]). LFTL has good scalability like DFTL and can adapt to various memory sizes. However, as introduced in Section II-B, the data buffer of LeaFTL occupies the memory space, and even worse the model-embedded GTD of LearnedFTL can't be kept in the small memory on UFS. In our experiments, the data buffer of LeaFTL is always 25% of the mapping cache. The model-embedded GTD of LearnedFTL is 1.5MiB for the emulated UFS. To make a fair comparison, we add a "No-reserve" SRAM assignment mode, in which all the 2MiB are assigned to save L2P mappings. Besides the unrealistic mode, the "Half-reserve" model applies to other FTLs, excluding LearnedFTL. The memory allocations are shown in Table III. The on-demand loading cache size of DFTL, TPFTL, and our LFTL is set to 2MiB and 1MiB for these two models, respectively.

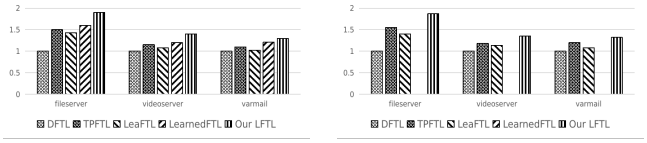
B. Microbench - FIO

We use the FIO benchmark [27] to evaluate the performance of sequential reads, random reads, sequential writes, and random writes for different FTL designs. Since the F2FS on



(a) No-reserve mode(2MiB). (b) Half-reserve mode (1MiB).

Fig. 6. The normalized throughput of FIO.



(a) No-reserve mode(2MiB). (b) Half-reserve mode (1MiB).

Fig. 7. The normalized throughput of Filebench.

TABLE IV
WORKLOADS FROM FILEBENCH.

Name	Description	Files	Threads	R/W
fileserver	Many large files with random writes	225000×128KiB	50	70/30
videoserver	Mostly sequential reads and writes	25×1GiB	48	20/80
varmail	Many small files with random writes	475000×16KiB	16	50/50

TABLE V
NUMBER OF ITERATIONS AND RATIO OF OUTLIERS.

	fileserver	videoserver	varmail
avg# of iterations	1.08	1.03	1.12
% of outliers	15.5%	11.2%	19.8%

mobile device We launch the tests after the storage device is warmed up. LearnedFTL relies on garbage collection to create accurate linear models. To investigate the performance in aged conditions when the device and/or file system perform garbage collection. We create a large file (20GiB) and repeatedly overwrite 4 KiB blocks randomly over the file until 60GiB data has been written. We use 512 KiB and 4 KiB I/O requests for sequential and random tests, respectively.

The evaluation results of no-reserve mode are shown in Figure 6(a). For random read, LFTL outperforms DFTL, TPFTL, LeaFTL, and LearnedFTL by 1.31×, 1.32×, 1.16×, and 1.08×, respectively. For sequential read, LearnedFTL outperforms others by 1.18×, 1.07×, 1.08×, and 1.05×, respectively. The evaluation results of half-reserve mode are shown in Figure 6(b). For random read, LFTL outperforms DFTL, TPFTL, LeaFTL, and LearnedFTL by 1.37×, 1.33×, 1.19×, and 1.08×, respectively. For sequential reads, LearnedFTL outperforms others by 1.19×, 1.06×, and 1.09×, respectively. The random read-throughput of learning-based FTLs is significantly higher than the DFTLs. Since the linear model covers a range of L2P mappings, more mappings can be found in memory, so as to provide a higher probability of cache hit. LFTL outperforms the other learning-based FTL since we overcome their shortages. For instance, the small data buffer limits the creation of long linear models in LeaFTL, and the CMT is too small to hold the L2P mappings of un-GCed pages. The performance of random writes is close to the sequential writes because 90% of the writes are sequential in F2FS. LFTL outperforms DFTL by 1.13× when the memory is assigned in half-reserve mode.

The throughput improvements of LFTL compared with the other FTLs are even more remarkable with tight memory budgets. First, LFTL doesn't consume memory space for accumulating sequential writes, it saves valuable memory for mapping cache. Second, LFTL unifies the management of linear models and point mapping entries in the mapping cache, they are loaded in memory on demand to efficiently utilize the memory space for hot mappings.

C. Application - Filebench

We select 3 typical workloads from Filebench [23] to make a comprehensive evaluation of LFTL. The descriptions of the workloads are listed in Table IV. Figure 7 shows the normalized throughput of different FTLs. In the no-reserve mode, LFTL outperforms other schemes by 1.07× to 1.9×. In the half-reserve mode, LFTL outperforms other schemes by 1.1× to 1.87×.

Overhead analysis. During the run of the workloads, we randomly sample 5% of the linear groups to analyze the overhead of the linear model creation. The overhead includes both time cost and memory cost. The time cost of the RANSAC linear regression is mainly dominated by the iterations. The memory cost correlates with the number of outliers. The sampled results are shown in Table V. The average number of iterations is very close to 1 and the outliers are less than 80%. The main computing overhead of learning-based FTL is sorting [13], which takes five times of training on an arm core. Fortunately, we don't need to sort the data, the time consumption of our linear model creation is acceptable. The number of outliers is less than 20%, and more than 80% of the L2P mappings are compacted in linear models.

VI. CONCLUSION

In this paper, we propose the LFTL, a learning-based FTL for mobile flash storage. Our observation on the LPA trace shows the opportunity to get the linear pattern from the orderly, LPA-aligned write stream. We use a space partitioning method to log the write stream in linear groups and adopt a RANSAC-based linear regression method to filter the random writes. The linear models and point mapping entries are managed together in the on-demand loading cache. By doing this, highly accurate linear models can be constructed regardless of memory limitation. The fully functional prototype of LFTL is implemented on FEMU, and the evaluation results of the FIO benchmark and filebench show that LFTL is more adaptable to memory-constrained storage devices than state-of-the-art learning-based approaches.

VII. ACKNOWLEDGEMENT

The work described in this paper is partially supported by the grants from the Research Grants Council of the Hong Kong Special Administrative Region, China (GRF 14219422, GRF 14202123), and the National Natural Science Foundation of China under Grant 62072333.

REFERENCES

[1] R. Chen, Y. Wang, J. Hu, D. Liu, Z. Shao, and Y. Guan, "Unified non-volatile memory and nand flash memory architecture in smartphones," in *The 20th Asia and South Pacific Design Automation Conference*. IEEE, 2015, pp. 340–345.

[2] B. Mao, J. Zhou, S. Wu, H. Jiang, X. Chen, and W. Yang, "Improving flash memory performance and reliability for smartphones with i/o deduplication," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 38, no. 6, pp. 1017–1027, 2018.

[3] J. Kim, S. Kim, J. Yun, and Y. Won, "Energy efficient io stack design for wearable device," in *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing*, 2019, pp. 2152–2159.

[4] A. Gupta, Y. Kim, and B. Urgaonkar, "Dftl: a flash translation layer employing demand-based selective caching of page-level address mappings," *Acm Sigplan Notices*, vol. 44, no. 3, pp. 229–240, 2009.

[5] Y. Yao, J. Fan, J. Zhou, X. Kong, and N. Gu, "Hdftl: An on-demand flash translation layer algorithm for hybrid solid state drives," *IEEE Transactions on Consumer Electronics*, vol. 67, no. 1, pp. 50–57, 2021.

[6] S. Jiang, L. Zhang, X. Yuan, H. Hu, and Y. Chen, "S-ftl: An efficient address translation for flash memory by exploiting spatial locality," in *2011 IEEE 27th Symposium on Mass Storage Systems and Technologies (MSST)*. IEEE, 2011, pp. 1–12.

[7] S. Lee, B. Lee, K. Koh, and H. Bahn, "A demand-based ftl scheme using dualistic approach on data blocks and translation blocks," in *2011 IEEE 17th International Conference on Embedded and Real-Time Computing Systems and Applications*, vol. 1. IEEE, 2011, pp. 167–176.

[8] J.-W. Hsieh, H.-Y. Lin, and D.-L. Yang, "Multi-channel architecture-based ftl for reliable and high-performance ssd," *IEEE Transactions on Computers*, vol. 63, no. 12, pp. 3079–3091, 2013.

[9] J.-Y. Hwang, S. Kim, D. Park, Y.-G. Song, J. Han, S. Choi, S. Cho, and Y. Won, "{ZMS}: Zone abstraction for mobile flash storage," in *2024 USENIX Annual Technical Conference (USENIX ATC 24)*, 2024, pp. 173–189.

[10] C. Wu, Q. Li, C. Ji, T.-W. Kuo, and C. J. Xue, "Boosting user experience via foreground-aware cache management in ufs mobile devices," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 11, pp. 3263–3275, 2020.

[11] J. No12, G. Lee, Y. Kim, and J. Jeong, "Fully harnessing the performance potential of dram-less mobile flash storage."

[12] J. Sun, S. Li, Y. Sun, C. Sun, D. Vucinic, and J. Huang, "Leaftl: A learning-based flash translation layer for solid-state drives," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, 2023, pp. 442–456.

[13] S. Wang, Z. Lin, S. Wu, H. Jiang, J. Zhang, and B. Mao, "Learnedftl: A learning-based page-level ftl for reducing double reads in flash-based ssds," in *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2024, pp. 616–629.

[14] C. Lee, D. Sim, J. Hwang, and S. Cho, "{F2FS}: A new file system for flash storage," in *13th USENIX Conference on File and Storage Technologies (FAST 15)*, 2015, pp. 273–286.

[15] K. G. Derpanis, "Overview of the ransac algorithm," *Image Rochester NY*, vol. 4, no. 1, pp. 2–3, 2010.

[16] H. Li, M. Hao, M. H. Tong, S. Sundararaman, M. Bjørling, and H. S. Gunawi, "The {CASE} of {FEMU}: Cheap, accurate, scalable and extensible flash emulator," in *16th USENIX Conference on File and Storage Technologies (FAST 18)*, 2018, pp. 83–90.

[17] "Jedec standard. zoned storage for universal flash storage (ufs)." pp. 220–5, 2022.

[18] T. Kraska, A. Beutel, E. H. Chi, J. Dean, and N. Polyzotis, "The case for learned index structures," in *Proceedings of the 2018 international conference on management of data*, 2018, pp. 489–504.

[19] Z. Sun, X. Zhou, and G. Li, "Learned index: A comprehensive experimental evaluation," *Proceedings of the VLDB Endowment*, vol. 16, no. 8, pp. 1992–2004, 2023.

[20] R. Marcus, E. Zhang, and T. Kraska, "Cdfshop: Exploring and optimizing learned index structures," in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, 2020, pp. 2789–2792.

[21] J. Ding, U. F. Minhas, J. Yu, C. Wang, J. Do, Y. Li, H. Zhang, B. Chandramouli, J. Gehrke, D. Kossmann *et al.*, "Alex: an updatable adaptive learned index," in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, 2020, pp. 969–984.

[22] A. Jens, "Block io tracing," *URL: git://git.kernel.dk/blktrace.git*, 2022.

[23] "Filebench," *URL: https://github.com/filebench/filebench*, 2021.

[24] Y. Zhou, F. Wu, P. Huang, X. He, C. Xie, and J. Zhou, "An efficient page-level ftl to optimize address translation in flash memory," in *Proceedings of the Tenth European Conference on Computer Systems*, 2015, pp. 1–16.

[25] Y. Zhou, Q. Wu, F. Wu, H. Jiang, J. Zhou, and C. Xie, "{Remap-SSD}: Safely and efficiently exploiting {SSD} address remapping to eliminate duplicate writes," in *19th USENIX Conference on File and Storage Technologies (FAST 21)*, 2021, pp. 187–202.

[26] K. Han, H. Gwak, D. Shin, and J. Hwang, "Zns+: Advanced zoned namespace interface for supporting in-storage zone compaction," in *15th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 21)*, 2021, pp. 147–162.

[27] A. Jens, "Flexible i/o tester," *URL: https://github.com/axboe/fio*, 2022.