Space-efficient FTL for Mobile Storage via Tiny Neural Nets

Ron Marcus
Technion
Haifa, Israel
marcus.ron@campus.technion.ac.il

Alon Rashelbach
Technion
Haifa, Israel
alonrs@campus.technion.ac.il

Ori Ben-Zur
Technion
Haifa, Israel
ori.b@campus.technion.ac.il

Pavel Lifshits
Technion
Haifa, Israel
pavel@ee.technion.ac.il

Mark Silberstein
Technion
Haifa, Israel
mark@ee.technion.ac.il

ABSTRACT

We present *RQFTL*, a demand-based FTL for mobile storage controllers that boosts the effective Logical-To-Physical (L2P) address translation cache capacity over state-of-the-art techniques. RQFTL stores a large part of the L2P cache in a compressed form, and employs a *learned data structure* called RQRMI that leverages tiny neural nets to quickly find the correct translation entry in the cache. RQFTL uses neural network inference for cache lookups, and rapidly retrains the neural nets to efficiently handle L2P cache updates. It is specifically optimized to achieve high coverage for scattered read accesses, making it suitable for popular read-skewed workloads such as mobile gaming.

We evaluate RQFTL on hours-long real-world I/O traces of popular modern mobile apps, including games, video editing, and social networking apps collected on Google Pixel 6a phone. We show that RQFTL outperforms all the state-of-the-art FTLs in these workloads, increasing the effective L2P cache capacity by over an order of magnitude compared to DFTL and up to 5× over the recent LeaFTL. As a result, it achieves 65%, and 25% lower miss rate compared to DFTL and LeaFTL respectively, under the same SRAM capacity, and allows reduction of the total SRAM capacity of a controller by about a third of that of LeaFTL.

SYSTOR '24, September 23–24, 2024, Virtual, Israel

ACM ISBN 979-8-4007-1181-7/24/09.
https://doi.org/10.1145/3688351.3689157

CCS CONCEPTS

• **Computer systems organization** → **Firmware**; • **Computing methodologies** → Supervised learning; • **Information systems** → *Unidimensional range search.*

KEYWORDS

SSD, FTL, Range Matching, Mobile Storage

1 INTRODUCTION

The storage demands of mobile systems are on the rise. Popular mobile phones already feature a TB-large SSD [46], and many apps, such as games, video editing, and media, use GBs of data on disk. Thus, maintaining high storage performance is essential for a satisfactory user experience.

Such workload scaling puts significant pressure on internal Flash Translation Layer (FTL) structures, and in particular, the Logical-To-Physical (L2P) address translation cache. The L2P translation table is itself stored on flash. Thus, to avoid accessing the flash twice on each I/O, FTLs use a memory-resident L2P translation cache that keeps the most frequently-used page mappings. In mobile storage, the L2P cache is particularly important not only for performance, but also for reducing the dynamic power consumption of the SSD, as it decreases the amount of reads of the L2P translations.

Many popular mobile apps feature read-skewed, mostly random I/O accesses scattered across a wide range of logical addresses. Our analysis of the three most popular I/O-heavy mobile games (§3) corroborates this observation. For example, the I/O accesses of Diablo are scattered over 120GB without any visible spatial locality, and this behavior does not change throughout the whole multi-hour gaming session (Figure 2). Such access pattern results in poor locality of L2P translation table accesses, and requires a large L2P cache to accommodate growing working sets.

Increasing the size of the L2P cache is quite costly. Mobile SSD controllers feature a small SRAM of about 512KB [21]

and only about a half of it is available for the L2P cache. At the same time, SRAM is known to be the dominant power and area consumer on a die, so adding more SRAM would have a sizable negative impact on the cost of the controller and its power consumption. For example, the area size of an ARC EM6 embedded processor used in popular SSD controllers is about 0.01 mm^2 [1] whereas 512KB-large SRAM block at 7nm technology is $0.28mm^2$ [27].

Another solution is a Host Performance Booster [21], where the L2P cache is stored in the phone's DRAM. However, this mechanism is no longer supported by the Android Linux Kernel due to a lack of community interest [8]. The successor technology, zoned UFS, has not been adopted so far either [9].

A preferable alternative is to reduce the L2P table's footprint by leveraging its internal structure. Specifically, prior works [18, 44] observed that the L2P table often contains contiguous strides of logical page addresses that are mapped to contiguous strides of physical pages. Thus, a more compact representation is to store one mapping per *range*. For example, the recent LeaFTL [44] demonstrated that such ranges are abundant in the enterprise SSD setting, and exploiting them in the FTL reduces its memory footprint.

However, applying range-based compaction in mobile storage controllers turned out to be tricky. We found that the *data structures* to allow efficient lookup and update of ranges incur prohibitively high space overheads. These overheads are less critical for enterprise-grade SSDs targeted by LeaFTL, as they keep the whole L2P table in a DRAM of several GBs. In contrast, for mobile storage controllers, almost *a half of the SRAM capacity* dedicated for the L2P cache is wasted on these data structures (see §3 for the detailed analysis).

We introduce *RQFTL*, a novel approach for range-based L2P compaction with a small memory footprint suitable for mobile storage. RQFTL improves the SRAM space utilization compared to LeaFTL, extending the effective L2P cache coverage thereby reducing translation miss rate in mobile workloads. Our key idea is to use a recent space-efficient *range-matching data structure*, RQRMI [39]. One of the unique aspects of RQRMI is its use of *tiny neural networks* to enable fast range matching while storing the ranges in a dense array. Thus, the space overheads of RQRMI are determined by the neural network size, which is independent of the L2P cache capacity and sums up to about 6KB.

The challenge, however, is that RQRMI does not support fast updates: its neural networks must be retrained to incorporate new mappings added to the cache, and the training may take a few milliseconds, which clearly cannot sustain a realistic L2P cache update rate. To solve this problem, we introduce a small *Short-Term Cache* to absorb all the cache insertions. This cache stores the mappings in a space-inefficient but small hash table. The newly cached ranges are periodically transferred into a large *Long-Term Cache* indexed using space-efficient RQRMI. Cache lookups start in the Long-Term Cache, and, if missed, continue to the Short-Term Cache. These two caches share the same SRAM space, which is dynamically partitioned among them depending on the access pattern.

RQFTL effectively serves most read I/O operations with the read-after-read reuse pattern from the Long-Term Cache. Instead, reads with the read-after-write reuse pattern are served from the Short-Term Cache. Since our target workloads are read-skewed, a large portion of the cache space is indexed using the space-efficient RQRMI data structure. The result is a logically unified L2P cache with low space overheads.

We implement RQFTL in a WiscSim [14] SSD simulator and thoroughly evaluate it using a variety of real-world block-I/O traces from several popular I/O-intensive mobile apps. For that purpose, we collect a large set of new mobile traces while running apps from popular categories, including video editing, social instant messaging, and several games, with over 24 hours of traces in total. We run the apps on a Google Pixel 6a mobile phone and record the traces using a small eBPF program without modifying the Android kernel. We put special emphasis on mobile games as they are the most downloaded type of mobile apps with the annual number of downloads higher than all other app categories combined [43].

On microbenchmarks RQFTL demonstrates up to 40× higher cache capacity than the popular DFTL [13] and up to 5× compared to LeaFTL [44]. It achieves 1.65× and 1.25× lower miss rate in the mobile application traces compared to them. In synthetic benchmarks that stress the L2P cache, RQFTL achieves up to 3.6× and 2× lower miss rate compared to DFTL and LeaFTL, with an overall latency speedup of 1.5× and 1.2×, respectively. Notably, these read performance benefits come without degrading the performance in write-intensive workloads. Last, RQFTL reduces the total L2P cache SRAM capacity *by a third* of that required by LeaFTL while achieving the same I/O performance. Importantly, we show that the computational and power overheads due to the RQRMI training are negligible.

2 BACKGROUND

SSDs consist of several *flash packages* which allow performing many read, program (write), and erase operations in parallel. Read and write operations are performed at a *page* granularity, typically 4KB in size. Yet, a page must be erased before it can be written. Erasure is performed at a *flash block*

[1] https://www.synopsys.com/dw/ipdir.php?ds=arc-em4-em6

SSD type	Memory type	Memory size	L2P cache
Enterprise	DRAM	1GB per 1TB of capacity	All entries
Consumer	SRAM	Hundreds of KB [21]	Hot Entries

Table 1: Properties of different SSD types.

granularity, where blocks are often groups of hundreds of pages [36].

Flash Translation Layer (FTL). Flash erasures are much slower than writes. Thus, data updates are performed out-of-place. The Flash Translation Layer (FTL) maintains dynamic mapping from logical addresses, called *Logical Page Numbers* (LPNs), to the corresponding physical addresses, called *Physical Page Numbers* (PPNs). The mappings are maintained in a *Logical to Physical* (L2P) table, which itself is stored on flash. To avoid two accesses to the flash for each I/O (first to translate LPN to PPN and then to the actual data page), the FTL caches the L2P table in a dedicated fast memory such as DRAM or SRAM.

Types of SSDs and L2P cache. Table 1 summarizes the main differences between the SSDs of different types. Enterprise-grade SSDs must satisfy strict tail latency guarantees, so they store their entire L2P map in a DRAM hosted on the SSD itself. Consumer-grade and mobile SSDs are DRAM-less. They are equipped with a small SRAM, which serves a *cache* for the most frequently used L2P mappings. These are fetched from the flash *on-demand* by the FTL. Hence, such FTLs are called *demand-based*. In the rest of the paper, we focus on such demand-based FTLs for mobile storage devices.

Mobile SSD overview. Figure 1a shows a high-level structure of a mobile SSD. The SRAM stores three main objects: a write buffer used to absorb writes before they reach the flash, the L2P cache, and a *Global Translation Directory* (GTD) which holds the location of the L2P mappings on the flash as explained below.

In flash, L2P mappings are stored in dedicated pages called *translation pages* (TPs). A typical TP holds 1K mappings. When the mappings within the TP are modified, the TP needs to be rewritten to a new PPN. The mapping between a TP LPN and its PPN is stored in the GTD.

Mappings could be performed at the granularity of pages, blocks, or some combination of the two [13, 26, 50]. We focus on page-based mapping, which is more popular for its efficiency, but comes at the cost of a larger memory footprint.

2.1 Range-matching using Neural Nets

Range-matching algorithm finds an integer interval (*range*) that contains an input integer out of a given set of non-overlapping ranges. For example, for the input 3 and the dataset $R_1 = [0, 4], R_2 = [7, 9]$, the algorithm produces R_1 as its output. Similarly, the input 6 produces NIL.

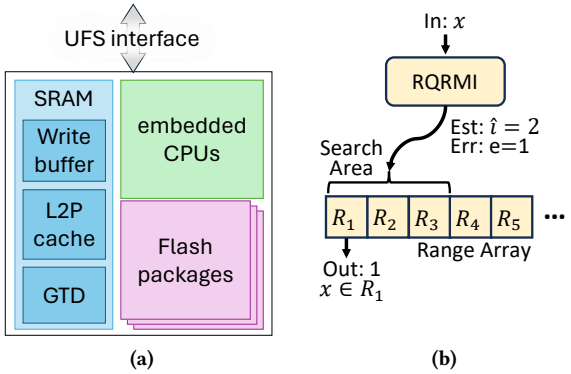

Figure 1: (a) Mobile SSD Overview. (b) RQRMI [39] lookup procedure.

This work uses range-matching to find the L2P mappings in the compressed L2P cache. Classical range-matching algorithms require traversing a tree that stores the ranges [47], but this approach does not scale well. Recently, a new range-matching technique called *Range-Query Recursive Model Index* (RQRMI) was shown to improve scalability and efficiency of the existing techniques. So far, RQRMI has been used in networking [39], but we apply it to L2P lookups in this work.

The RQRMI model is a mixture of experts (MoE), comprising a hierarchy of tiny fully connected neural nets (Multilayer Perceptrons), each with one input, eight hidden nodes with ReLU activation, and one output [39]. The model is first trained to learn the distribution of sorted, non-overlapping integer ranges $S = \{R_1, R_2, \ldots, R_n\}$ in memory. We call S a *Range Array*. A query (Figure 1b) is performed in two steps: (Step 1) The input x is fed into the RQRMI model which outputs the *estimate* of the index of the matching range in S ($\hat{i}$) and an upper bound on the prediction error computed during training ($e < \log n$); (Step2), a secondary search over a subarray of S at $[\hat{i} - e, \hat{i} + e]$ outputs the index (if there is a match). In the example, $\hat{i} = 2, e = 1$, and the matching range is at $i = 1$.

RQRMI is the only learned index data structure that supports range-matching and guarantees small memory footprint and lookup correctness. The time to training a model depends on the RQRMI size and the end-goal lookup latency [40]. For the models used in this work, the training takes a few milliseconds.

3 MOTIVATION

3.1 Analysis of I/O-heavy mobile apps

Modern mobile apps pose significant performance and capacity requirements for storage. In this paper, we put a particular emphasis on mobile games. Games are by far the most popular segment of apps: they are the most downloaded type of apps exceeding all other app categories combined [43]. At the same time, many popular games are quite I/O heavy, consuming hundreds of GBs of space on the SSD, and featuring multi-GB working sets.

We collected several hours-long gaming I/O traces on a modern Google Pixel 6a phone (we provide more details on the trace collection methodology in §7.1). Figure 2 visualizes the access pattern over time for three representative games. We can see that the I/O operations are dominated by many reads scattered over random LPNs. Moreover, adjacent accesses often span large address ranges, so their L2P mappings are located in different Translation Pages (TPs). We believe this behavior stems from loading many small assets throughout the game. At the same time, we assume the I/O access latency is critical for a pleasant gaming experience.

This kind of access pattern is particularly challenging to serve at low latency as it is not L2P cache-friendly, in particular under stringent L2P space constraints. Specifically, each miss in the cache results in doubling the number of flash accesses on each read (i.e., one access to the page with the LPN-to-PPN mapping, and the other to the actual data). Not only L2P misses affect the read latency, but they also increase the dynamic power of the storage device accordingly.

3.2 Why not host-side L2P caching

Host DRAM is sometimes used to increase the memory for FTL [21, 42]. Specifically, *Host Performance Booster* (HPB) dedicates a few hundred MBs from the host's DRAM to store the FTL data structures, including the L2P cache. It also uses the host CPU to execute FTL operations [21].

Nevertheless, HPB has disadvantages: it uses CPU cycles and DRAM capacity that would otherwise be available to user applications. Moreover, for energy-sensitive devices, the use of the host CPU is not an option. Last, HPB support has been *removed* from the Linux kernel [8].

ZUFS (zoned UFS) [16] is expected to be HPB's successor. However, it also puts pressure on the host resources, and some smartphone makers do not intend to support it [9].

3.3 FTL compaction using ranges

Techniques such as SFTL [18] and LeaFTL [44] take advantage of the fact that multiple logically contiguous pages are often written sequentially into physically contiguous pages on the disk. Thus, a single *compressed entry* can encode multiple mappings in a single *range*. Such a compact representation of the mappings helps increase the number of L2P mappings in the cache.

For example, for a single write spanning three logical pages, the SSD allocates three contiguous LPNs `0x0001`, `0x0002`, `0x0003` and three contiguous PPNs `0x1001`, `0x1002`, `0x1003`. Thus, instead of saving three mapping entries, a single range mapping entry `0x0001-0x0003 -> 0x1001` suffices.

Range-based compaction depends on the availability of ranges in the LPN and PPN address space. The former is predicated on the low fragmentation of the LPN space. Fortunately, low fragmentation is beneficial for the SSD performance in general, and it is one of the design goals of modern file systems, therefore we assume that it is indeed a common case. On the other hand, PPNs are mostly allocated and written sequentially to the same flash block by design, therefore the same-write allocations are usually contiguous in the PPN space. Arguably, some PPN ranges get eventually broken during garbage collection, but as most of the storage space is written rarely, the vast majority of ranges will remain intact. In practice, LPNs and PPNs are allocated when the data is being written, so the range sizes are dictated primarily by the *size of the SSD write requests*.

The recent LeaFTL project explored the opportunity to compact L2P mappings using ranges, with an average range size of about 4-5 pages in real-world enterprise storage traces. We make similar observations for mobile apps, i.e., the average range size in Genshin Impact game trace varies between 25 and 75 pages (see §7.3).

3.4 Range compaction for mobile FTL

LeaFTL demonstrated impressive performance benefits of range-based compaction compared to other popular FTL techniques such as DFTL [13] and SFTL [18]. However, it focused primarily on enterprise SSDs with abundant DRAM. Thus, the compaction of the L2P table in DRAM allowed allocating more DRAM space for the *data cache*, leading to improved end-to-end performance.

In contrast to enterprise SSDs, mobile storage SSDs do not have enough space for data caching. Thus, the primary benefit of range-based compaction is to increase the L2P cache coverage, and as a result, reduce read latency and power consumption.

We, therefore, attempted to apply LeaFTL in mobile setting. Unfortunately, we find that LeaFTL suffers from certain space overheads that have been relatively negligible in enterprise SSDs but make LeaFTL less suitable for mobile devices. We discuss these next.

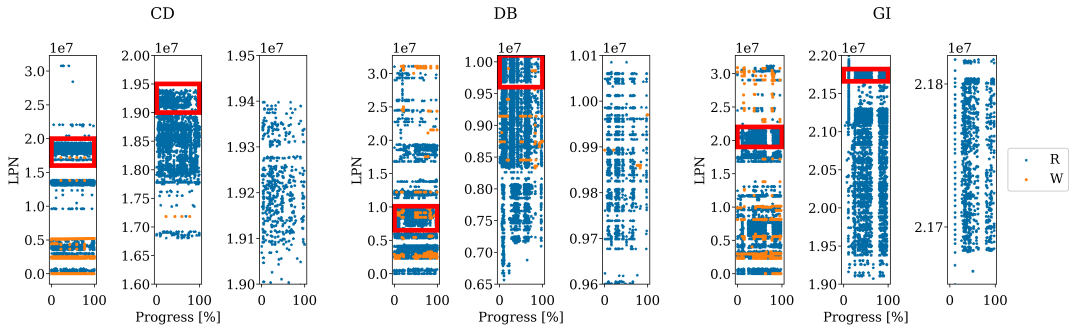

Figure 2: The I/O access pattern (LPNs accessed over time) of three popular games (see §7.1 for the collection methodology): Call of Duty (CD), Diablo (DB), and Genshin Impact (GI). The first 100K I/O operations are shown. The leftmost subplot shows the entire address span, and the subplots to the right show the zoomed-in LPN range in the red box. The blue dots are read commands, and the orange are write commands.

3.4.1 Space overheads of LeaFTL. We now give a brief background on LeaFTL and then discuss its overheads. Note that SFTL [18] suffers from similar issues.

LeaFTL overview. LeaFTL [44] uses segmented linear regression for recognizing patterns in its LPN to PPN mappings. The compressed segments are *sorted* and stored in multiple layers of a log data structure maintained for each Translation Page (TP). Given an LPN, LeaFTL first searches the RAM for the location of the TP that contains the mapping to the relevant PPN. The TP's mappings are stored in a multi-layer log structure similar to a Log-Sort-Merge-Tree to support fast updates. Specifically, the TP's log is scanned layer by layer, from top to bottom, until the most up-to-date segment that contains the LPN is found. Since the segments are sorted in each layer, the in-layer search is performed via binary search. Layers of the multi-layer log are periodically flattened to reduce the number of layers to scan.

Unaccounted space overheads. Unlike uncompressed TPs which can be stored in a simple array with fixed-size entries, LeaFTL stores compressed TPs whose size depends on the compressibility, i.e., the number of the ranges in that page. Therefore, when compressed TPs are used, a hash table is required for indexing the locations in memory of cached TPs. However, a hash table incurs over-provisioning overheads to avoid collisions. Further, the multi-layer log data structure, which is an essential part of LeaFTL, requires a sorted data structure with fast insertions such as an AVL tree. Such a data structure relies on dynamic memory allocation, implying that pointers must be accounted as part of its memory footprint. We are not aware of any other suitable data structure that is more compact than AVL.

We calculated the actual memory footprint of LeaFTL including the overheads as follows. We assume the best-case scenario of highly compressible mappings, with a segment size of 256. So a translation page is represented by 4 segments with a memory footprint of $4 \times 9 = 36$ Bytes (each segment is 9 Bytes: 4 Bytes for PPN, 2 Bytes for LPN offset from translation page start, 2 Bytes for segment slope, 1 Byte for segment length).

We note that originally, LeaFTL used translation pages with only 256 LPNs and a segment memory footprint of 8 Bytes (The LPN offset was 1 Byte). However, we set the TP size of 1024 LPNs, because it is better for LeaFTL as it reduces the hash table overhead (4× fewer hash table entries).

Now, we consider the space requirements of all the other data structures. First, each range is stored in a node of an AVL tree. Each node has an overhead of 5 Bytes: 2 pointers of 2 bytes each, and another byte for the AVL-related information. This sums up to the overhead of $4 \times 5 = 20$ Bytes per translation page.

Secondly, we consider the hash table memory overhead. Each entry in the hash table has a size of 5 Bytes (3 Bytes for the translation page number, and another 2 Bytes to store the translation page location in SRAM). The hash table size is provisioned to suffice for the maximum number of cached translation pages to avoid rehashing upon cache insertion, with a moderate load factor of 0.5 [34]. This sums up to an overhead of $\frac{1}{0.5} \times 5 = 10$ Bytes per translation page.

Together, the AVL and hash-table overheads are 30 Bytes per TP. This translates to $\frac{30}{30+36} = 45\%$ of the memory. This is a pure overhead: only 55% of the SRAM capacity is used to store the actual mappings.

3.5 Opportunity: range-matching data structures

The recent RQRMI range-matching data structure may reduce the above space overheads significantly, improving the effective L2P cache capacity. As discussed in §2.1, RQRMI uses a small learned model based on tiny neural nets to

enable fast retrieval of matching ranges from a sorted array, with strict performance guarantees. The model space overheads depend on the total array size and the desired lookup performance. In our case, the model may occupy as little as 6KB, a negligible overhead for a 256KB cache, and a significant improvement over LeaFTL.

Compelled by these observations, we design the L2P cache using the RQRMI model as its main data structure.

4 DESIGN

4.1 Considerations

Using RQRMI as the basis for the L2P cache management is challenging because, fundamentally, RQRMI is an immutable data structure. Thus, adding or removing a range requires retraining RQRMI neural networks on the updated array of ranges. Doing so on every cache update is not feasible.

There have been several attempts to enable updates to learned data structures [7, 10, 29]. There are three main approaches: (1) using Log-structured Merge (LSM) Trees [30], (2) leaving unoccupied gaps in the sorted data array to afford insertion of new entries [7], and (3) absorbing updates in an auxiliary updatable data structure. All these approaches periodically retrain the model on the updated data.

We decided against the first two approaches. LSM trees have significant space and management overheads such as space amplification [30] that we strive to avoid. The second approach also wastes space and results in slower lookup unless the model is retrained. This is because the insertions effectively make the model less precise with respect to the new data, so the model outputs more loose predictions and results in a longer search. Further, this approach is not suitable for caches: there could be a burst of insertions that might exhaust the gap between two adjacent array entries, so accommodating new entries would require evicting older entries in the same area of the array. However, such an eviction policy would be far from optimal, thereby affecting cache performance.

Thus, we adopt the idea of maintaining a single *updatable* data structure that absorbs the updates. We periodically merge its content into the RQRMI range array, and retrain the model.

We note that the updatable and immutable parts of the cache compete for the same memory space. The amount of memory allocated for each part depends on the access pattern and may have a significant effect on the cache performance. We discuss how RQFTL deals with dynamic memory partitioning in §4.7.

4.2 Overview

RQFTL L2P cache comprises two parts: an updatable section called *Short-Term Cache (STC)* and an immutable section called *Long-Term Cache (LTC)*. The former may use any existing cache, whereas the latter uses RQRMI. All the L2P updates are absorbed in the STC and then periodically transferred to the LTC. The lookup first checks the LTC, and if missed, continues to the STC.

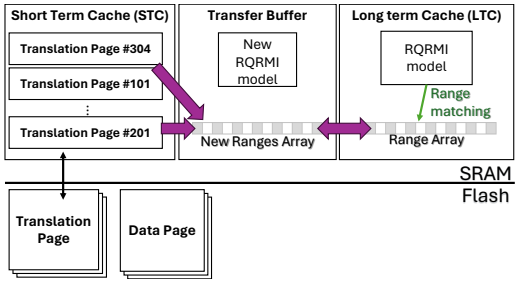

Figure 3: The RQFTL L2P cache design has two parts: Short-Term Cache (STC) and Long-Term Cache (LTC). All updates are first inserted into the STC and periodically merged into the LTC using the transfer buffer, followed by an RQRMI training.

During the transfer of the ranges from the STC to the LTC (§4.5), the merged array that aggregates the contents of both caches is temporarily stored in a *transfer buffer*. This array is used to train a new RQRMI model. The retrained model that incorporates the new ranges is placed in LTC, atomically replacing the old one.

The old LTC mappings stay available during the transfer process. The training uses a dedicated embedded core, so the lookup latency is not affected. The core can be repurposed for other tasks when write intensity is low (§7.4).

We now describe the cache structure in detail.

4.3 Short-Term Cache (STC)

The STC may use any existing FTL. We chose SFTL [18] for its small memory footprint and simplicity. A single cache line in the STC is a single Translation Page (TP). The cache is managed in a hashtable.

Each cache line keeps an AVL tree of ranges and a bitmap that encodes the ranges' stating LPN and length efficiently.

We note that the size of the STC changes dynamically depending on the STC-to-LTC partitioning §4.7. However, since the STC is designed to store the minority of the mappings in the L2P cache, the overall overhead of its data structures is relatively small. Each cache line in the STC has a dirty bit reflecting that the translation page on flash must be updated.

Caching individual ranges. SSD writes create new L2P mappings as they are performed out-of-place. Typically, the relevant translation page is fetched from the flash to be merged with the updated mappings and thus is cached in its entirety in the L2P cache. However, the STC only stores

the individual updates, which we call *patches*, and the actual merge occurs only upon patch eviction. This technique further reduces the memory requirements of the STC.

STC cache policy. The small size of the STC allows us to maintain a precise LRU replacement policy at the granularity of a single translation page. Note that STC is managed independently of the replacement policy of the LTC. The LRU list is updated on every lookup.

Insertion. We define two types of insertions: read-insertion and write-insertion, due to SSD reads and writes respectively. Upon read-insertion, a whole translation page is fetched from the flash and stored in the STC cache line. The cache evicts one translation page entry according to the LRU policy.

Upon write-insertion, if a translation page is already present in the STC, a new patch for that range is inserted, and the translation page is marked dirty. In addition, if the LPNs of the inserted range are found in any ranges already cached in the LTC, those ranges are invalidated in the LTC to ensure that the lookups return the most recent updated range (recall that upon the lookup the LTC is queried first).

Eviction. A translation page without dirty ranges is removed from the cache without further actions. A dirty translation page is written to flash. If the STC holds only a subset of the mappings in a translation page, the translation page is first read from flash to merge the updates and only then is written back.

4.4 Long-Term Cache (LTC)

The LTC is maintained in an RQRMI data structure. Specifically, RQFTL stores LPN ranges in a *range array*, sorted by the starting LPN. Each range is represented using 9 Bytes: 4 bytes for the starting LPN, 4 bytes for the starting PPN, and 1 byte for the range length. Ranges are limited to at most 255 pages as we found that larger ranges are rarely created. An RQRMI model is trained on that range array (§2.1).

LTC cache policy. As the size of the LTC is quite large, we approximate the LRU behavior using a well-known one-handed CLOCK algorithm [3]. Each range has an "accessed" bit, updated upon a cache hit. When ranges need to be evicted, we run through the range array, marking for eviction the ranges with the cleared access bit, otherwise clearing it.

Range invalidation. The LTC maintains a validity bitmap of the size of the ranges array. As mentioned earlier, when a range is inserted into the STC, a prior version of the mappings in that range may already have been cached in one or more ranges in the LTC. These LTC ranges must be invalidated. To do so, the relevant ranges are found by performing RQRMI inference for all the newly inserted LPNs. The matching ranges are marked as invalid and eventually evicted.

Eviction. Evictions are performed as part of the process that merges between the STC and the LTC (see below). Note that there are no dirty ranges in the LTC as they are all handled by the STC, so the evicted ranges are simply removed from the range array.

4.5 Transferring from the STC to LTC

New mappings in the STC are periodically transferred to the LTC. The maximum frequency at which these transfers can be performed is upper-bounded by the maximum training rate of the RQRMI model. It does not depend on the number of new ranges, but rather, on the size of the range array. A transfer process starts once the previous one has finished unless there are no new mappings to be transferred. We show the impact of the transfer rate on the performance in §7.3.

The transfer process comprises the following steps:

(1) **Selection of the STC ranges.** Unmarked STC ranges are moved to a temporary array in the *transfer buffer* (Figure 3) and get marked by a dedicated bit per TP in the STC. Updates that occur during the transfer process clear the mark bit for their relevant TPs.
(2) **LTC eviction.** Invalid LTC entries and valid entries of TPs marked in the STC are removed from the LTC. If the transfer buffer contains more ranges than the LTC can accommodate, the LTC eviction process (see above) is invoked.
(3) **Merge sort.** The LTC entries are merged with the ones in the transfer buffer. The temporary array is then sorted.
(4) **Training.** The RQRMI model is trained on the temporary range array.
(5) **LTC update.** The new model and the updated array are atomically replaced with the ones in the LTC. The access bits of all new ranges are raised to avoid premature eviction.

4.6 Putting it all together

The following describes a step-by-step example of RQFTL L2P cache operation (illustrated in Figure 4):

At `t0`, the STC is empty and the LTC contains the mappings of two translation pages (TPs): #99 and #101. At `t1`, a read of an LPN in TP#100 causes a miss in both LTC and STC; therefore, TP #100 in its entirety is fetched from flash into the STC. At `t2`, the new mappings from TP #100 are transferred from the STC to the LTC. At `t3`, a new RQRMI model is trained on the new mappings. The *version column* (v) in the LTC is changed from 0 to 1. TP#100 is held by both the STC and the LTC and will eventually be evicted from the STC (`t4`). At `t5`, an LPN in TP#100 is accessed and causes an LTC hit. At `t6`, write events to LPNs 100-120 and 200-300 cause a patch in the STC and an invalidation of the respective ranges in the LTC. At `t7`, TP#100 gets evicted from the STC. At `t8`, LPN 800 from TP#100 gets accessed and hits the LTC

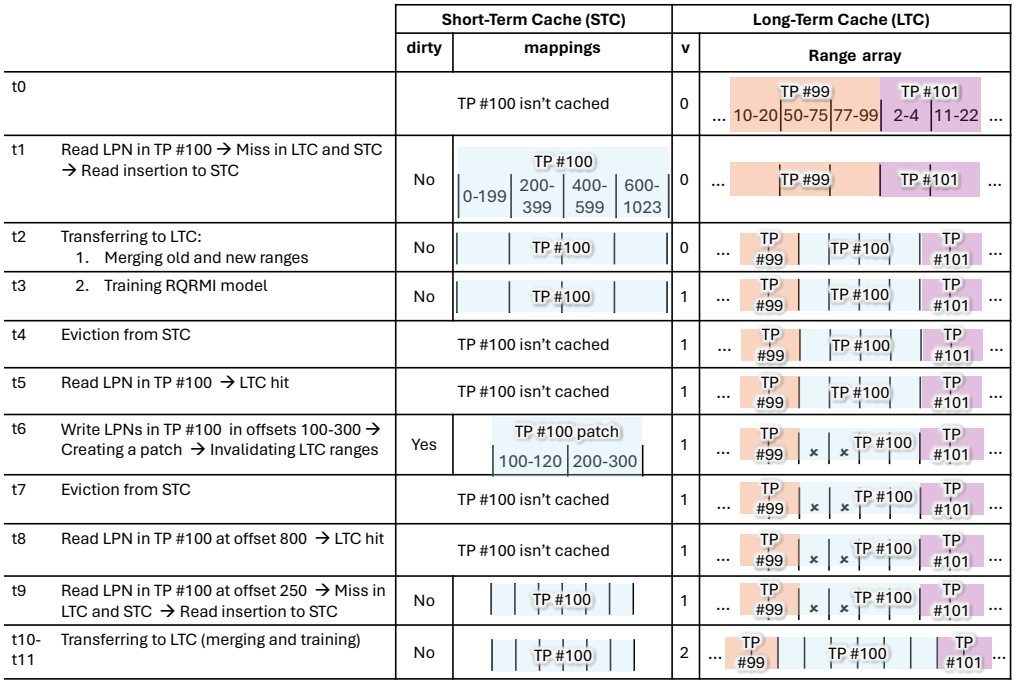

Figure 4: RQFTL example (see §4.6)

since it is marked as valid. At `t9`, a read of LPN 250 in TP#100 occurs and causes a miss in both the LTC (marked as invalid) and the STC (not covered). This leads to *read-insertion* and the entire TP is fetched into the STC. Last, at `t10-t11` the new mappings are transferred from the STC to the LTC. This results in purging the invalid entries in the LTC, merging the rest with the STC, and training a new RQRMI model.

4.7 Optimizations

Dynamic memory partitioning between the STC and LTC. The STC and LTC share the same memory space: increasing the size of one comes at the expense of the size of the other.

The partitioning primarily depends on the access pattern. For example, a read-only workload with a large working set would benefit from a larger LTC, whereas a write-intensive workload with read-after-write reuse would work better with a larger STC. Another factor that affects the partitioning is the RQRMI training rate, which determines the freshness of the LTC. We discuss the impact of the training rate in §5.

In contrast, bad partitioning might lead to a severe performance drop. Since updates are performed in the STC it needs to be large enough to avoid thrashing, otherwise leading to slow SSD performance and shorter lifespan.

These observations highlight the inherently workload-dependent nature of the partitioning. Therefore, RQFTL takes an elastic approach where the STC size is determined by the intensity of L2P updates. RQFTL keeps a counter of the number of updated translation pages occurred between the LTC-to-STC transfers. Thus, the counter offers an estimate of the necessary space in the STC. We use a short-term running average of the history of the counter values to avoid abrupt changes. The remaining space is used for the LTC. The size of the STC is limited to 512 translation pages we found empirically to work well with the partitioning heuristic.

Note that adjusting the partition size causes eviction from the cache part whose size gets reduced.

Reducing the transfer buffer size. The L2P cache must operate correctly during the STC-to-LTC transfers. Therefore, while training the new RQRMI model on the transfer buffer, the LTC with its old RQRMI model and its respective range array must be kept intact. This way, the effective memory footprint of the LTC is doubled, which is unacceptable.

To reduce the space requirements of the transfer buffer, we randomly partition the original range array into multiple disjoint arrays we call *fractions*. Each fraction holds a subset of the ranges in LTC, and is indexed with its own (smaller) RQRMI model. Thus, for F fractions, the size of the transfer buffer is $1/F+1$ of the total SRAM capacity. We use a simple hash function to find the fraction that holds the mappings of a specific LPN. The STC-to-LTC transfers are performed for each fraction at a time in a round-robin manner.

Coalescing multiple ranges. RQFTL uses the write buffer to aggregate writes. When the buffer is full, it is written to

the flash. Before writing, the data pages are sorted according to their linear addresses. This process allows us to coalesce adjacent ranges thus allowing the creation of larger ranges.

4.8 End-to-end memory footprint analysis

The translation pages in the STC are indexed by a hash table, with a load factor of 0.5, and entry size of 10B, exactly as we assumed for the LeaFTL analysis (§3). However, since we limit the STC to 512 translation pages, the hash table overhead for RQFTL is at most $512 \times 10B = 5$ KB.

The space overhead in the LTC has two factors: the transfer buffer, and the RQRMI models. Assuming $F = 12$, the size of the transfer buffer is $1/13$ of the LTC size, which is less than 20KB for a 256KB L2P cache. The 12 RQRMI models have a memory footprint of 6KB in total. Together, the LTC space overhead is no more than 26KB. Thus, the total space overhead of RQFTL is at most *31KB*, and it is independent of the total capacity of the L2P cache.

5 DISCUSSION

Read-optimized FTL design. Our L2P design deliberately optimizes for read-skewed workloads with short read I/Os, low spatial locality, and large reuse distance, which is common in mobile I/O-heavy apps that motivated this work (§3). This is because RQFTL allocates the largest part of the cache to the LTC that serves SSD reads, whereas the range updates due to SSD writes are stored in the small STC. As a result, RQFTL performs well in a workload with read reuse. Yet, it is on par with the state-of-the-art in workloads with read-after-write reuse pattern, as corroborated by the evaluation.

Training time and end-to-end cache behavior. Note that the transfer of new mappings from the STC to the LTC is not synchronized with the cache metadata management. Such a design ensures that the training process is not in the critical path of the cache updates.

However, the cache performance may suffer due to slow RQRMI training. In one extreme, if the training is slow and STC-to-LTC transfers happen infrequently, the LTC will not reflect the actual working set of the cache, whereas the STC will likely start thrashing. On the other extreme, if the training time is short, the STC and the LTC are no longer separate.

We evaluate the impact of the training time on the end-to-end performance in §7.3.

Crash consistency. RQFTL relies on the standard capacitor-based approach to crash consistency of mappings. When the power goes down, the dirty translation pages are written to flash. Importantly, only the STC contains dirty mappings, so the LTC can be shut down without consuming power.

Other core FTL functions. RQFTL focuses only on the L2P cache and does not introduce changes to the core FTL functions, such as a Garbage Collector.

6 IMPLEMENTATION

We implement RQFTL in WiscSim [14], a well-established trace-driven SSD simulator. We use the same version of the simulator as the most recent LeaFTL [44] project, thereby allowing fair comparison with LeaFTL.

Our implementation follows the description in §4, and is enclosed in the L2P cache logic.

We inherit the extensions introduced by LeaFTL [44], and further enhance the simulator as follows [2]:

PPN allocation scheme. Upon write buffer eviction, the simulator allocates PPNs for the flushed data pages. The PPNs are allocated sequentially, to allow the creation of ranges. However, in the original simulator, sequential PPNs were stored in the same flash block, on the same flash die. Since flash dies can only perform one operation at a time, such an allocation policy did not exploit the inter-die parallelism and harmed the write performance.

To remedy this, we changed the constant mappings between PPNs and the location in the physical flash hierarchy. In the new mapping, sequential PPNs are striped into different flash dies. In other words, instead of numbering pages from the same block as sequential PPNs, we number pages from the same stripe as sequential PPNs. This concept was presented in [48]. It is worth noting that the new numbering scheme is static and does not introduce additional lookups.

Grouping of translation pages. Recall that the GTD holds the physical locations of the translation pages. Consequently, the number of consecutive TPs per GTD entry dictates its total size. Therefore, it is common to *group* several TPs together to reduce the GTD size. For example, a 1TB SSD with 4K pages supports an overall of 256K TPs. Thus, if the TPs are not grouped, the GTD would take 1MB of SRAM (256K TPs with 4B per PPN). Instead, we use groups of eight TPs so the GTD size is reduced by 8. Naturally, every rewrite of a TP requires modification to the whole group.

RQRMI. We use the open-source implementation of RQRMI inference [39] and re-implement the fast training technique [40] without SIMD acceleration.

7 EVALUATION

We seek to answer the following questions:

(1) How does RQFTL compare to state-of-the-art FTLs on different workloads?
(2) What is the effect of the design parameters on the performance?
(3) What are the resource overheads?

[2] We intend to open-source our changes upon acceptance.

7.1 Methodology

Baselines. We compare RQFTL with three page-based on-demand FTLs: DFTL [13] that is commonly used in practice, SFTL [18] that employs per-TP range compression, and the most recent LeaFTL [44] that uses advanced L2P compression. We implement SFTL and DFTL in WiscSim, and use the public version of LeaFTL [44]. For a fair comparison, we include the memory footprint overheads (§3) for all FTLs. We consider a variation of DFTL which caches entire translation pages instead of individual LPNs.

Configuration of the simulated SSD. Unless stated otherwise, we configure the capacity to 1TB and allocate 128KB for the write buffer, 128KB for the Global Translation Directory, and 256KB for the L2P cache. Hence, the total SRAM size is 512KB. These numbers match the typical sizes of mobile UFS SSDs [20]. We use 4KB data pages. In terms of parallel resources, the number of planes in the SSD is configured to 512, which is similar to [12] considering the SSD capacity. The flash read latency is 200μs per page, and the flash write latency is 1200μs per page. We set the queue depth to 32 as configured in the Pixel 6a phone we use for collecting the traces. We configure the L2P cache line size as a single translation page in all FTLs. We note that LeaFTL originally used a 4X smaller cache line size, which is impractical under the memory constraints of DRAM-less SSD because the cache index overhead would be too big. Note that garbage collection wasn't triggered for any of the FTLs.

Real-world traces. We collect I/O traces from several popular mobile applications using a Pixel 6a mobile phone [3]. For the trace collection, we instrumented the Android Linux Kernel using a special eBPF function that recorded all the accesses to the storage devices on the phone, on the block level. We made sure HPB is disabled.

The main characteristics of all the collected traces are shown in Table 2. Among the apps we used, we specifically included a subset of the most popular games, with the largest storage footprint that could fit on our phone: Pubg, Call of Duty, Diablo Immortal, and Genshin Impact [4].

In addition, we collect the traces of several representative storage-heavy applications from other popular app categories such as social media (Telegram), picture collections (F-Stop gallery), and video editing (YouCut video editor).

The traces were collected while actively using the apps for several hours. As a result, the traces also include the I/O activity of other processes concurrently running on the phone in the background. To ensure that the trace represents the app being used, we explicitly checked that the vast majority (over 95%) of the I/O requests originate from that app.

[3]The traces and the tools used to collect them will be published upon acceptance.

[4]According to https://www.thegamer.com/.

We note that the phone has been extensively used before the experiments, exhausting the SSD capacity and subsequently erasing data. Thus, we believe the traces feature a realistic fragmentation of the LPN space on a disk. This is further corroborated by the large span of accessed LPNs all over the LPN address space of a 120GB SSD.

For completeness, we also evaluate RQFTL on the enterprise workloads taken from Microsoft Research Cambridge (MSR) and Florida International University (FIU) [23, 33].

Trace initialization and replay. Each trace comprises two parts: write I/O during the installation of the app on the phone, and the actual app execution trace. We first replay the installation I/O trace to initialize the FTL to create correct L2P mappings, without measuring system performance. Next, we replay the execution I/O trace and report its performance here. The simulator automatically pre-writes all the LPNs that are read but do not have the matching write in the trace.

Unless stated otherwise, we obtain the performance results by replaying the traces without their I/O timestamps, i.e., maintaining full SSD queues at maximum I/O rate. This is a common practice in prior works [44].

RQRMI configuration. All RQFTL end-to-end measurements use the RQRMI performance characteristics collected from a single ARM Cortex A72 core @ 2GHz using the real-world mobile traces mentioned above. We use small two-layer RQRMI models with 4 NNs and train them using 16K samples. The overall memory footprint used by RQRMI models is about 6KB. RQRMI training latency was set according to our experiments §7.4.

The LTC fraction count (F) was configured to 12.

7.2 End-to-end Results

Mobile traces. Figure 5 shows the end-to-end performance of DFTL, SFTL and RQFTL. To put these results in context, we also show the performance of an ideal L2P cache with unlimited size. RQFTL achieves an average end-to-end miss rate reduction of 65%, 24%, and 25% compared to DFTL, SFTL, and LeaFTL, respectively, which translates into 15%, 4%, and 4% average read latency speedup. In specific applications, such as Call of Duty (CD), RQFTL achieves 54% lower miss rate compared to the second best SFTL, which results in 9% improvement in read latency.

When taking writes into account, the overall I/O latency improvement is lower. In addition, in write-skewed applications, SFTL and LeaFTL performance is on par or slightly better than RQFTL. This is an acceptable result, as RQFTL is specifically optimized for read-skewed workloads, but does not cause performance degradation otherwise.

Server-grade traces. We run the server-grade storage traces and observe similar performance as LeaFTL. More details can be found in §11.

	Length [cmds]	TP Reuse Distance [cmds]		Write req size [Pages]		Read req size [Pages]		LPN span [GB]	Read Ratio	Workset Size [GB]
		Avg.	SD	Avg.	SD	Avg.	SD			
Call of Duty (CD)	2496029	2020.7	9799.3	28.9	60.7	11.3	28.6	117	0.91	5.1
Diablo (DL)	1898152	2417.2	12736.6	4.5	12.6	5.3	17.2	119	0.82	4.3
Genshin Impact (GI)	1022753	1202.6	5181.2	42	56.4	11.8	24.9	118.9	0.94	2.8
Pubg (PB)	658321	737	4031.9	12.4	31.2	6.3	32	119	0.48	2
Slideshow (SS)	4609251	16086.3	51323.2	38.1	31.6	4.3	8.4	119	0.94	1.9
Telegram (TG)	1538672	2304.5	14228.9	4.4	20.1	15.3	61.2	119	0.24	3.9
YouCut (YC)	8167619	2105	13791	80.9	58.2	29.7	7.6	72.4	0.99	10.1

Table 2: Statistics of real-world mobile app traces collected from the Pixel 6a phone.

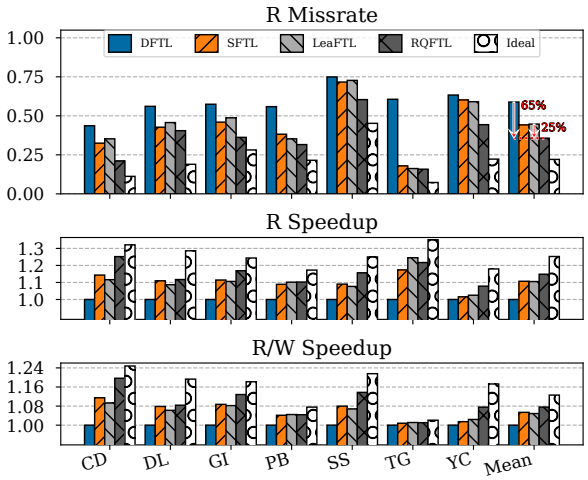

Figure 5: Performance on mobile workloads: Call of Duty (CD), Diablo (DB), Genshin Impact (GI), Pubg (PB), Slideshow (SS), Telegram (TG), YouCut (YC).

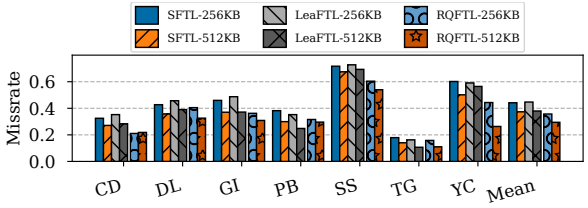

Figure 6: Miss rate of RQFTL vs. LeaFTL and SFTL with the cache sizes of 256KB and 512KB.

SRAM savings. To quantity the SRAM saving of RQFTL over the alternatives, we measured how much more SRAM would they need to get the same miss rate as RQFTL. As seen in Fig. 6, LeaFTL needs 2× more SRAM to get the same average performance. For some traces (Call of Duty and You Cut), RQFTL gets even higher performance than LeaFTL with the L2P twice as large. Also, if all FTLs get the enlarged SRAM, RQFTL gets the lowest miss rate.

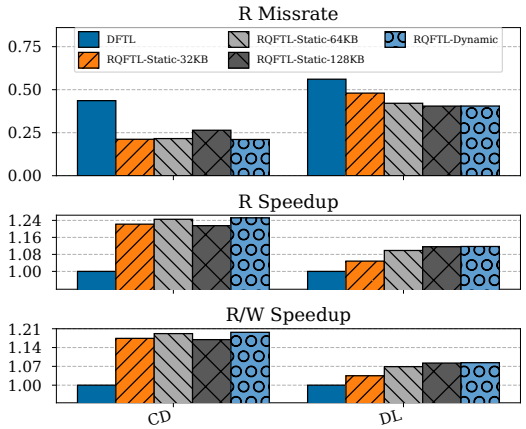

Figure 7: Performance of different SRAM partitioning methods in RQFTL. Three sizes of the STC (32KB, 64KB, and 128KB) for the static method are evaluated.

For the total SRAM savings, we should include the GTD and the write buffer. So all of them together with the L2P cache occupy 512KB for RQFTL, and 768KB for LeaFTL. Thus RQFTL saves about one-third of the SRAM capacity needed to achieve the same performance. If we include the caches of the on-die CPUs (64KB), then the savings reduce to 30%.

7.3 Detailed Analysis

Dynamic memory partitioning. Figure 7 compares the performance of SRAM partitioning methods - dynamic (§4.7) and static, on two representative workloads (Call of Duty and Diablo Immortal games). It shows that there is no static STC size that matches both workloads. For Call of Duty, STC size of 64KB archives the best miss rate and R/W latency; while for Diablo Immortal, 128KB is best. The dynamic method gets the highest performance on all traces, without manual configuration.

Synthetic benchmarks. We simulate the behavior of a system under stress using synthetic workloads that consist of two phases: a *preconditioning* phase, which issues many

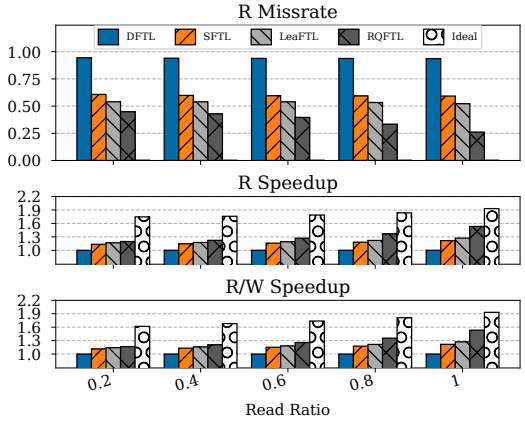

Figure 8: Performance on synthetic workloads for different reads/writes ratios. Ratio=1 - Read-only.

	CD	DL	GI	PB	SS	TG	YC
RQFTL ($\times 10^6$ LPNs)	1.4	0.3	0.8	0.3	1.9	0.4	1.8
RQFTL/DFTL (rel. cap.)	26.2	7.5	17.2	13.6	38.3	17.5	30.3
RQFTL/SFTL (rel. cap.)	2.9	1.4	2.0	1.9	3.6	1.9	2.7
RQFTL/LeaFTL (rel. cap.)	3.7	2.0	2.3	1.4	4.7	1.6	3.9

Table 3: The effective L2P capacity for mobile traces. RQFTL row: actual LPN capacity. Other rows: relative capacity of RQFTL compared to other methods.

random write commands and results in a sequential PPN page allocation, followed by a *test* phase, which issues a configurable mix of read and write commands. We chose such a workload because it is conceptually similar to the contiguous-write-random-read access pattern in real traces.

Workload generation is as follows. We define an *LPN span* that contains logical page addresses (e.g., 0-16GB), and randomly select a set of pages on which the workload would operate. This set of pages, called the *workset*, is not necessarily contiguous and may be smaller than the LPN span (e.g., 4GB). All writes are of the same size (e.g., 32 pages) and are aligned accordingly.

Figure 8 depicts the end-to-end performance of RQFTL compared to the alternatives, as a function of the read ratio (the total pages read divided by the total pages accessed). RQFTL achieves up to 3.6×, 2.7×, and 2× lower miss rate compared to DFTL, SFTL, and LeaFTL, respectively. These results demonstrate that RQFTL can adapt to write-intensive workloads and highlight its superior performance in read-only scenarios.

Cache capacity. We compared the average total number of LPNs cached by each FTL on mobile app traces. Table 3 shows that RQFTL may effectively cache over a million translations in a 256KB L2P cache, which is significantly higher than any other FTL approach.

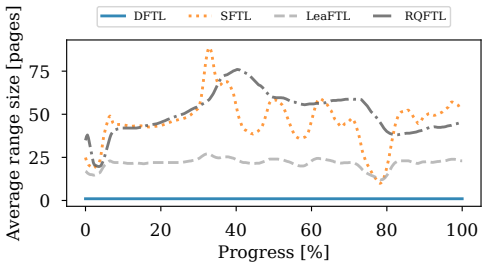

Figure 9: The average size of ranges in the L2P cache over time during Genshin Impact execution.

Range size. Figure 9 depicts the average range size in the L2P cache over the execution of a representative application (Genshin Impact). We notice that the size varies between 25 pages to 75 pages. Thus, encoding the range size with 1 Byte is indeed enough. RQFTL and SFTL get higher range sizes than LeaFTL by merging adjacent ranges where possible.

Write amplification. Write amplification (WAF) is defined as the ratio between the total flash writes to the actual writes issued by the user. WAF is expected to be high in cases where writes exhibit high miss rate. For example, reoccurring writes to the same translation page might cause several flash write events in case the translation page was evicted between the writes. Thanks to the elastic design of RQFTL, the write miss rate experienced in RQFTL is on-par with that of LeaFTL, and the WAF is similar to all other FTL schemes.

Additional experiments. RQFTL shows little sensitivity to the flash access latency, and also handles large LPN spans better than the alternatives. See §11 for the graphs.

7.4 RQRMI performance and power overheads

Lookup latency. We measure the RQRMI inference latency on the same ARM core and obtain the average of 170ns, less than 0.1% of the time it takes to read a flash page.

Training time. We measure the CDF of the RQRMI training time on the real-world mobile traces using a single ARM Cortex A72 core @ 2GHz. We use this as a performance estimate for training on a power-efficient core on a storage controller. We note that the same code takes about 1ms to run on the modern Intel AlderLake X86 core. The median latency is 7.5ms, and the 90^{th} percentile is 8.8ms. These values dictate the maximum rate at which the LTC will be updated.

The training latency is lower than [39] because the range array is smaller, and the model is smaller too.

Next, we evaluate the maximum RQFTL performance if the training would take zero time (RQFTL* in Figure 10). These results indicate a relatively small miss rate reduction for

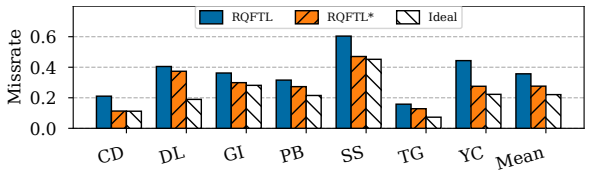

Figure 10: The effect of a hypothetical zero RQRMI training latency (marked as RQFTL*) over the read miss rate, compared to RQFTL.

RQFTL* (8 miss rate percentage points on average), implying that the current training rate is acceptable.

Power and execution overheads. By taking timestamps from I/O traces into account, we find that RQRMI training is a rare event. In all of the mobile-grade traces, training takes place in 1% to 3% of the execution time. Therefore, the dedicated core can be landed for other usages most of the time. Alternatively, the training does not require any additional cores and can be executed on one of the device's existing CPU cores.

8 RELATED WORK

Address translation for SSDs. A variety of works target the issue of address translation in flash devices [1, 2, 6, 13, 15, 18, 19, 26, 35, 37, 44]. We focus on the most relevant for our work. DFTL [13] suggests to use demand-based address translation. It caches the least recently used mappings by storing the accessed translation pages in RAM. SFTL [18] introduces mapping compression by leveraging ranges. It uses bitmaps and *popcnt* operations to encode ranges in a TP. LeaFTL [44] is the state-of-the-art when it comes to employing range compression for address translation in flash devices. To the best of our knowledge, no other scheme addresses the unique constraints of modern mobile storage by using space-efficient indexing for ranges.

LeaFTL vs. RQFTL. Both LeaFTL and RQFTL use machine learning techniques to improve the L2P cache performance. However, whereas LeaFTL uses segmented linear regression to identify complex L2P patterns, it stores the ranges in a translation page and suffers from poor space efficiency. RQFTL, on the other hand, uses RQRMI model-based data structure to perform range matching, which enables it to store ranges from all the translation pages in a large dense array. Thus, in the context of DRAM-less storage, RQFTL makes better use of the memory resources. Furthermore, while LeaFTL's cache line is the entire translation page, RQFTL can cache subsets of translation pages, preventing the wasteful caching of unused mappings.

LearnedFTL. LearnedFTL uses segmented linear regression model, similarly to LeaFTL. It cleverly minimizes flash reads due to inaccurate model predictions, by using a bitmap filter. However, it assumes that all the models are cached in RAM (as part of the GTD), which is not feasible for DRAM-less mobile drives with a large capacity.

Machine learning for storage. Various works use ML optimizations for storage, such as finding the best SSD hardware configuration for a given workload [28], reducing garbage collection overhead [49], cache space reallocation [51], and cache replacement policy [41]. Yet, none use neural networks as a core mechanism for address translation in FTLs.

Learned indices. Learned data structures have been applied to databases, key-value stores, virtual software switches, and networking hardware [5, 7, 10, 22, 24, 25, 29, 38, 40, 45], and show performance benefits by trading memory accesses for ML inference computations. To the best of our knowledge, RQFTL is the first to apply such data structures in FTLs.

Mobile storage trace collection. Many works deal with storage trace collection for different applications. Some analyze FS-level traces [11, 17], others deal with deduplication [31, 32]. [4, 52] analyze the properties of mobile I/O using short traces. We recorded long traces of mobile storage-intensive applications at the block level to analyze the application behavior in the context of the L2P cache.

9 CONCLUSION

This work presents RQFTL, a new technique that employs tiny neural networks for boosting the performance of the address translation caches in mobile storage devices. RQFTL improves the space-efficiency of the existing techniques and enables better utilization of the limited SRAM space available in mobile storage controllers. We show that RQFTL outperforms the state-of-the-art on a variety of traces of real I/O-heavy mobile apps.

10 ACKNOWLEDGEMENTS

We thank the reviewers for their helpful comments and feedback. We thank Daniel and Benjamin Silberstein for their help in collecting traces. We gratefully acknowledge the generous support of the Israeli Science Foundation (Grant 1998/22).

REFERENCES

[1] 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 *File and Storage Technologies (FAST)*.

[2] 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.

[3] Fernando J Corbato et al. 1968. A paging experiment with the multics system. *Defense Technical Information Center* (1968).

[4] Jace Courville and Feng Chen. 2016. Understanding storage I/O behaviors of mobile applications. In *2016 32nd Symposium on Mass Storage Systems and Technologies (MSST)*. IEEE, 1–11.

[5] Yifan Dai, Yien Xu, Aishwarya Ganesan, Ramnatthan Alagappan, Brian Kroth, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2020. From WiscKey to Bourbon: A Learned Index for Log-Structured Merge Trees. In *USENIX Operating Systems Design and Implementation (OSDI)*.

[6] Niv Dayan, Philippe Bonnet, and Stratos Idreos. 2016. GeckoFTL: Scalable flash translation techniques for very large flash devices. In *International Conference on Management of Data*.

[7] Jialin Ding, Umar Farooq Minhas, Jia Yu, Chi Wang, Jaeyoung Do, Yinan Li, Hantian Zhang, Badrish Chandramouli, Johannes Gehrke, Donald Kossmann, David B. Lomet, and Tim Kraska. 2020. ALEX: An Updatable Adaptive Learned Index. In *ACM Management of Data (SIGMOD)*.

[8] Discussions and development of Linux SCSI subsystem. 2023. Remove HPB support. https://patchwork.kernel.org/project/linux-scsi/patch/20230719165758.2787573-1-bvanassche@acm.org/.

[9] embedded.com by AspenCore. 2024. Micron's new UFS 4.0 modules shrink dimensions and add functionality. https://www.embedded.com/microns-new-ufs-4-0-modules-shrink-dimensions-and-add-functionality/.

[10] Paolo Ferragina and Giorgio Vinciguerra. 2020. The PGM-index: a fully-dynamic compressed learned index with provable worst-case bounds. *Proc. VLDB Endow.* 13, 8 (2020), 1162–1175.

[11] Roy Friedman and David Sainz. 2016. File system usage in android mobile phones. In *Proceedings of the 9th ACM International on Systems and Storage Conference*. 1–11.

[12] Congming Gao, Liang Shi, Chun Jason Xue, Cheng Ji, Jun Yang, and Youtao Zhang. 2019. Parallel all the time: Plane level parallelism exploration for high performance SSDs. In *Mass Storage Systems and Technologies (MSST)*. IEEE, 172–184.

[13] Aayush Gupta, Youngjae Kim, and Bhuvan Urgaonkar. 2009. DFTL: a flash translation layer employing demand-based selective caching of page-level address mappings. In *ASPLOS*.

[14] Jun He, Sudarsun Kannan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2017. The Unwritten Contract of Solid State Drives. In *EuroSys*.

[15] Jian Huang, Anirudh Badam, Moinuddin K Qureshi, and Karsten Schwan. 2015. Unified address translation for memory-mapped SSDs with flashmap. In *International Symposium on Computer Architecture*.

[16] JEDEC. 2023. Zoned Storage for UFS. https://www.jedec.org/standards-documents/docs/jesd220-5.

[17] Cheng Ji, Riwei Pan, Li-Pin Chang, Liang Shi, Zongwei Zhu, Yu Liang, Tei-Wei Kuo, and Chun Jason Xue. 2020. Inspection and characterization of app file usage in mobile devices. *ACM Storage (TOS)* 16, 4 (2020), 1–25.

[18] 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 *IEEE Mass Storage Systems and Technologies (MSST)*.

[19] 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 Embedded Computing Systems (TECS)* 9, 4 (2010), 1–41.

[20] Jung-Hoon Kim, Sang-Hoon Kim, and Jin-Soo Kim. 2018. Utilizing subpage programming to prolong the lifetime of embedded NAND flash-based storage. *Consumer Electronics* 64, 1 (2018), 101–109.

[21] Yoona Kim, Inhyuk Choi, Juhyung Park, Jaeheon Lee, Sungjin Lee, and Jihong Kim. 2023. Integrated Host-SSD Mapping Table Management for Improving User Experience of Smartphones. In *File and Storage Technologies (FAST)*.

[22] Andreas Kipf, Ryan Marcus, Alexander van Renen, Mihail Stoian, Alfons Kemper, Tim Kraska, and Thomas Neumann. 2020. RadixSpline: a single-pass learned index. In *ACM Management of Data (SIGMOD)*.

[23] Ricardo Koller and Raju Rangaswami. 2010. I/O Deduplication: Utilizing Content Similarity to Improve I/O Performance. *ACM Trans. Storage* 6, 3 (2010), 1–26.

[24] Tim Kraska, Mohammad Alizadeh, Alex Beutel, Ed H Chi, Jialin Ding, Ani Kristo, Guillaume Leclerc, Samuel Madden, Hongzi Mao, and Vikram Nathan. 2019. SageDB: A learned database system. In *Innovative Data Systems Research (CIDR)*.

[25] Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, and Neoklis Polyzotis. 2018. The Case for Learned Index Structures. In *ACM Management of Data (SIGMOD)*.

[26] 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 *ACM international conference on Embedded software*.

[27] T. Lakshmi and M. Kamaraju. 2022. A Review on SRAM Memory Design Using FinFET Technology. *International Journal of System Dynamics Applications* 11 (01 2022), 1–21. https://doi.org/10.4018/IJSDA.302665

[28] Daixuan Li, Jinghan Sun, and Jian Huang. 2023. Learning to Drive Software-Defined Solid-State Drives. In *IEEE/ACM Microarchitecture*.

[29] Pengfei Li, Yu Hua, Pengfei Zuo, and Jingnan Jia. 2019. A Scalable Learned Index Scheme in Storage Systems. *CoRR* abs/1905.06256 (2019).

[30] Chen Luo and Michael J Carey. 2020. LSM-based storage techniques: a survey. *The VLDB Journal* 29, 1 (2020), 393–418.

[31] Bo Mao, Suzhen Wu, Hong Jiang, Xiao Chen, and Weijian Yang. 2017. Content-aware trace collection and I/O deduplication for smartphones. In *Proc. 33rd Int. Conf. Massive Storage Syst. Technol.(MSST)*. 1–8.

[32] Bo Mao, Jindong Zhou, Suzhen Wu, Hong Jiang, Xiao Chen, and Weijian Yang. 2018. Improving flash memory performance and reliability for smartphones with I/O deduplication. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 38, 6 (2018), 1017–1027.

[33] Dushyanth Narayanan, Austin Donnelly, and Antony I. T. Rowstron. 2008. Write off-loading: Practical power management for enterprise storage. *ACM Storage* 4, 3 (2008), 10:1–10:23.

[34] Rasmus Pagh and Flemming Friche Rodler. 2004. Cuckoo hashing. *Journal of Algorithms* 51, 2 (2004), 122–144.

[35] 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 Embedded Computing Systems (TECS)* 7, 4 (2008), 1–23.

[36] Seon-Yeong Park, Euiseong Seo, Ji-Yong Shin, Seungryoul Maeng, and Joonwon Lee. 2010. Exploiting Internal Parallelism of Flash-based SSDs. *IEEE Comput. Archit. Lett.* 9, 1 (2010), 9–12.

[37] Zhiwei Qin, Yi Wang, Duo Liu, and Zili Shao. 2010. Demand-based block-level address mapping in large-scale NAND flash storage systems. In *IEEE/ACM/IFIP hardware/software codesign and system synthesis*.

[38] Alon Rashelbach, Igor de Paula, and Mark Silberstein. 2023. NeuroLPM - Scaling Longest Prefix Match Hardware with Neural Networks. In *IEEE/ACM Microarchitecture (MICRO)*.

[39] Alon Rashelbach, Ori Rottenstreich, and Mark Silberstein. 2020. A Computational Approach to Packet Classification. In *ACM Data Communication (SIGCOMM)*.

[40] Alon Rashelbach, Ori Rottenstreich, and Mark Silberstein. 2022. Scaling Open vSwitch with a Computational Cache. In *USENIX Networked Systems Design and Implementation (NSDI)*.

[41] 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 *File and Storage Technologies (FAST)*.

[42] Sungyong Seo, Youngjin Cho, Youngkwang Yoo, Otae Bae, Jaegeun Park, Heehyun Nam, Sunmi Lee, Yongmyung Lee, Seungdo Chae, Moonsang Kwon, Jin-Hyeok Choi, Sangyeun Cho, Jaeheon Jeong, and Duckhyun Chang. 2016. Design and implementation of a mobile storage leveraging the DRAM interface. In *IEEE High Performance Computer Architecture (HPCA)*.

[43] Statista. 2023. Number of mobile app downloads worldwide from 2019 to 2027, by segment. https://www.statista.com/forecasts/1262881/mobile-app-download-worldwide-by-segment. Accessed: Jan 2024.

[44] 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 *ASPLOS*.

[45] Chuzhe Tang, Youyun Wang, Zhiyuan Dong, Gansen Hu, Zhaoguo Wang, Minjie Wang, and Haibo Chen. 2020. XIndex: a scalable learned index for multicore data storage. In *ACM Principles and Practice of Parallel Programming (PPoPP)*.

[46] TheVerge. 2021. Apple's iPhone 13 Pro is the first iPhone with 1TB of storage. https://www.theverge.com/2021/9/14/22673600/iphone-13-1-tera-storage-memory-price-apple.

[47] Cormen Thomas H, E Charles, Rivest Ronald L, Stein Clifford, et al. 2009. Introduction to Algorithms.

[48] Shengzhe Wang, Zihang Lin, Suzhen Wu, Hong Jiang, Jie Zhang, and Bo Mao. 2023. LearnedFTL: A Learning-based Page-level FTL for Improving Random Reads in Flash-based SSDs. *arXiv preprint arXiv:2303.13226* (2023).

[49] Qingsong Wei, Cheng Chen, Mingdi Xue, and Jun Yang. 2015. Z-MAP: A zone-based flash translation layer with workload classification for solid-state drive. *ACM Storage (TOS)* 11, 1 (2015), 1–33.

[50] Chin-Hsien Wu, Dong-Yong Wu, Hong-Ming Chou, and Che-An Cheng. 2017. Rethink the design of flash translation layers in a component-based view. *IEEE Access* 5 (2017), 12895–12912.

[51] Jianpeng Zhang, Mingwei Lin, Yubiao Pan, and Zeshui Xu. 2023. Crftl: cache reallocation-based page-level flash translation layer for smartphones. *IEEE Consumer Electronics* (2023).

[52] Deng Zhou, Wen Pan, Wei Wang, and Tao Xie. 2015. I/O characteristics of smartphone applications and their implications for eMMC design. In *2015 IEEE International Symposium on Workload Characterization*. IEEE, 12–21.

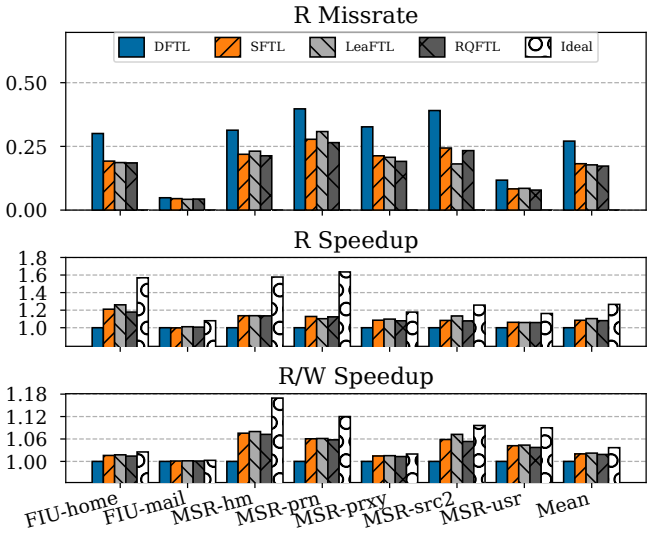

Figure 11: Performance on server-grade workloads.

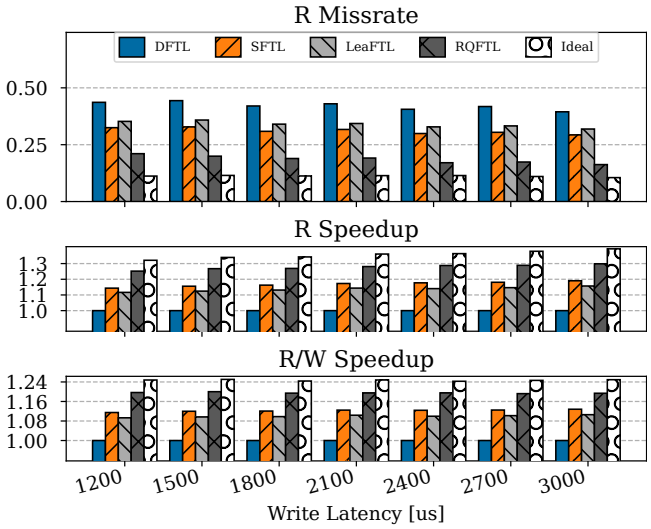

Figure 12: Performance under different flash write latency settings for the representative trace Call of Duty.

11 APPENDIX

Server-grade traces. Figure 11 depicts the end-to-end results on the server-grade workloads often used for FTL evaluation [23, 33]. The results show relatively similar performance of all the FTLs that use range compaction.

Note that the performance of LeaFTL reported here differs from the one published in [44], because of the DRAM-less setup of our system. In particular, mobile environments lack data caches aimed at reducing flash I/O operations as they would be ineffective given the little available SRAM space. Moreover, the small write buffer size limits sophisticated L2P pattern recognition techniques [44].

Insensitivity to flash latencies. Different SSDs (with different technology, manufacturer, form factor, etc.) exhibit different flash latencies. Figure 12 shows that RQFTL speedups are almost insensitive to the flash write latency, for the representative trace Call of Duty. For example, when varying the flash write latency from 1.2ms to 3ms, the read speedup of RQFTL over the second-best FTL (SFTL), only slightly varies: from 9.5% to 9.1%. Similarly, the experiments show RQFTL speedups are insensitive to read latency, between 150us and 200us.

Performance with a large LPN span. Figure 13 demonstrates that RQFTL adjusts to the *LPN span* better than any other technique. This property stems from its ability to maintain sparse mappings in range granularity rather than at the granularity of a whole translation page.

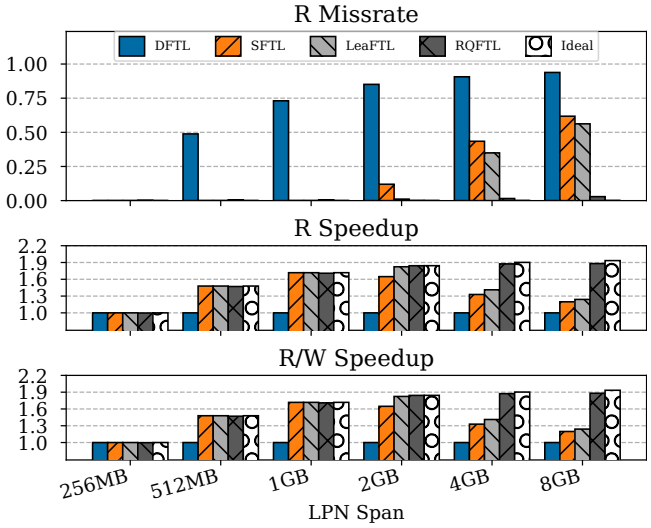

Figure 13: End-to-end results for synthetic workloads under varying the possible logical page number range. The Ideal FTL and RQFTL miss rate bars are barely visible