HyTorC: Hybrid Address Translation for SSDs Supporting Compression

YU ZHANG, Tianjin University, Huawei Technologies and Tsinghua University, Beijing, China
RENHAI CHEN, Tianjin University and Huawei Technology, Tianjin, China
GONG ZHANG, Huawei Technologies Co Ltd, Shenzhen, China
PENG WANG, Huawei Technology, Hong Kong, China
YAO XIN, Huawei Technology, Hong Kong, China
HUANG KEJI, Huawei Technology, Hong Kong, China
ANDRÉ BRINKMANN, Institute of Computer Science, Johannes Gutenberg-Universität Mainz, Mainz, Germany

High-capacity solid-state drives (SSDs) with expected capacities of one PByte and more will address cloud storage and archiving environments previously dominated by magnetic disks. These new applications are very cost-sensitive, so unnecessary overhead must be reduced as much as possible without sacrificing the performance advantages of SSDs.

An expensive component within a scale-up SSD is the on-device memory to map host logical page numbers to flash pages. This article, therefore, proposes HyTorC, which builds on the idea of S-FTL to represent sequentially stored logical pages using bitmaps instead of providing one entry per logical page. HyTorC extends this idea by introducing, for the first time in an FTL, compression of these bitmaps using run-length encoding, and by investigating the effects of background scrubbing to realign randomly written pages into contiguous runs. This background scrubbing allows HyTorC to keep large portions of the mapping table in block mapping mode, further reducing the memory footprint.

HyTorC retains the flexibility of the page mapping scheme and supports compression of logical blocks within the SSD, allowing multiple compressed logical pages to be stored within a single physical page. HyTorC is fully implemented in an open-channel SSD. Our tests show that HyTorC can reduce memory consumption by an average of 98.9% over standard page mapping, 95.6% over the DFTL scheme, 87.3% over S-FTL, and 56.5% over the learned index-based approach LeaFTL for the Alibaba Cloud block traces, and by 98.6% over standard page mapping, 95.5% over the DFTL scheme, 84.1% over S-FTL, and 61.5% over LeaFTL for the Microsoft Research Cambridge traces. HyTorC focuses on the memory footprint of the FTL and not on performance. However, the performance evaluation shows that HyTorC achieves similar performance compared to page mapping and FTLs based on learned indexes.

Authors' Contact Information: Yu Zhang, Tianjin University, Huawei Technologies and Tsinghua University, Beijing, China;; e-mail: rain_zhang@tju.edu.cn; Renhai Chen (corresponding author), Tianjin University and Huawei Technology, Tianjin, China; e-mail: chenrenhai@huawei.com; Gong Zhang, Huawei Technologies Co Ltd, Shenzhen, Guangdong, China; e-mail: nicholas.zhang@huawei.com; Peng Wang, Huawei Technology, Hong Kong, Hong Kong, China; e-mail: wangpeng423@huawei.com; Yao Xin, Huawei Technology, Hong Kong, Hong Kong, China; e-mail: yao.xin1@huawei.com; Huang Keji, Huawei Technology, Hong Kong, Hong Kong, China; e-mail: huangkeji@huawei.com; André Brinkmann (corresponding author), Institute of Computer Science, Johannes Gutenberg-Universität Mainz, Mainz, Rheinland-Pfalz, Germany; e-mail: brinkman@uni-mainz.de.

ACM 1553-3077/2026/03-ART14
https://doi.org/10.1145/3767335

CCS Concepts: • **Information systems** → **Information storage systems**; **Information storage technologies**; **Storage class memory**; **Flash memory**;

Additional Key Words and Phrases: Flash translation layer, data structures, Cloud based storage

ACM Reference Format:
Yu Zhang, Renhai Chen, Gong Zhang, Peng Wang, Yao Xin, Huang Keji, and André Brinkmann. 2026. HyTorC: Hybrid Address Translation for SSDs Supporting Compression. *ACM Trans. Storage* 22, 2, Article 14 (March 2026), 31 pages. https://doi.org/10.1145/3767335

1 Introduction

Flash-based **solid state drives** (**SSDs**) have replaced magnetic **hard disk drives** (**HDDs**) in many applications, such as mobile computing or as fast server storage. High-capacity SSDs, with expected capacities of one PByte and more, will also address cost-sensitive cloud storage and archiving environments previously dominated by HDDs. For these new application areas, unnecessary overhead must be reduced as much as possible without compromising the performance advantages of SSDs.

The most expensive component inside a scale-up SSD is the flash storage itself. New generations of SSDs, therefore, integrate a compression unit inside the controller that detects and removes redundancies within a data block [6, 18, 28, 43, 45]. This allows, in the simplest case, to store multiple logical pages within a single physical page on the SSD.

The second most expensive component of an SSD is its main memory, which is required, for example, to store mapping information to translate **logical page numbers** (**LPNs**) used by the host computer to **physical page numbers** (**PPNs**) on the SSD. The size of the corresponding mapping information grows with the number of logical pages, so growing SSD capacities and data compression inside SSDs increase the pressure on it.

Most modern SSDs apply a strategy called *page mapping* to translate between logical and physical numbers. Page mapping stores one entry per logical page in main memory, so the size of this page map grows with both the number of physical pages provided by the SSD and the compression rate. Therefore, as the price of flash memory falls faster than the price of main memory, and as the number of logical pages increases due to compression techniques, the relative cost of page mapping within an SSD also increases. For example, the price difference between memory and SSDs increased from 8× in 2010 to 42× in 2023 [25], even without considering compression inside SSDs.

Page mapping has been adopted by modern SSDs because more coarse-grained approaches, such as simple block mapping and hybrid approaches [17, 19], provide limited flexibility in where a page can be stored and incur too much performance overhead for the potential cost savings.

Figure 1 shows the memory requirements for address translation of a 1 PByte physical capacity SSD for different mapping strategies. To derive these results, we scaled the memory requirements for the subset of Alibaba cloud traces [20] used in our evaluation (see Table 1) for different address translation approaches to a capacity of 1 PByte.

Applying standard page mapping to 4 kByte physical pages requires nearly 2 TBytes of memory, even without further data compression. This means that, when applying page mapping, the cost for DRAM inside the SSD is in the same order as the cost for the flash storage. Caching only parts of the page table, as introduced by the **demand-based FTL** (**DFTL**) strategy [5], can reduce the memory footprint to 626 GBytes. However, caching either introduces delays for random access workloads or requires additional fast storage devices [7].

It is, therefore, important to rethink address translation techniques to reduce their memory size. An important insight has been that only one physical address needs to be stored for each set of sequentially stored logical pages [9, 33, 36, 44]. S-FTL was the first strategy to take advantage of

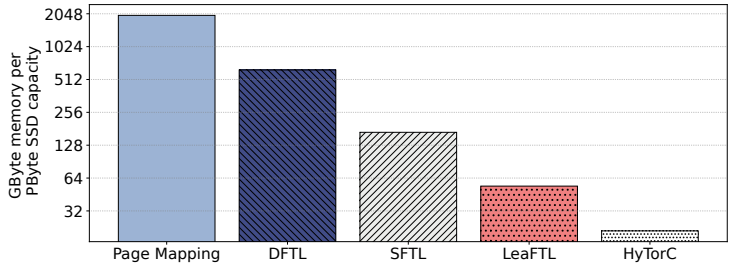


Fig. 1. Memory capacity required to map a 1 PByte SSD.

this insight, using bitmaps to encode sequentially stored logical pages, and reducing the memory requirement in our workload mix to 168 GBytes [9]. These sequential runs are also used by learned indexes [33, 36] such as LeaFTL [33], which is able to further reduce the memory requirement to 54 GBytes. Existing address translation algorithms can therefore already reduce the cost of DRAM for address translation to a few percent of the overall SSD cost. However, in a market with small profit margins, further reducing these costs can provide a significant competitive advantage, especially as the price difference between DRAM and flash memory further increases.

This article proposes the ***Hy**brid Address **T**ranslation f**o**r SSDs supporting **C**ompression* (HyTorC) algorithm, which significantly reduces mapping metadata by combining the advantage of page mapping, that each logical page can be stored at arbitrary physical locations, with an extremely compact metadata representation for sequentially stored logical pages, and a background scrubbing process to sequentially realign logical pages. Compared to the most memory efficient existing address translation algorithm, HyTorC is able to further reduce memory requirements by a factor of two, requiring only 21 GBytes of memory to fully map a 1 PByte SSD for our workload mix.

The **first contribution** of this article extends the insight that only one physical address needs to be stored for sequentially stored logical pages (see, e.g., [9, 33, 36, 44]). HyTorC for the first time couples the bitmap approach of S-FTL with an additional run-length encoding to compress the metadata needed to find boundaries between sequential runs. We also show that this run-length encoding has a negligible impact on the lookup performance of the SSD, even for the low-end CPUs used in today's SSDs.

The **second contribution** takes advantage of the fact that the lifespan of SSDs increases with their capacity. This is, compared to smaller SSDs, due to the reduction of daily overwrites at the same write bandwidth and the reduction in write amplification, as cold data in these SSDs implicitly increases the overprovisioning available for hot data [21]. Combined with internal SSD bandwidth growing faster than external SSD bandwidth, it becomes possible to support more **garbage collection** (**GC**) cycles or data reorganizations without impacting overall performance or SSD lifespan. HyTorC, therefore, adds a scrubbing process that realigns cold data to store it as sequentially as possible. Scrubbing allows HyTorC to store more page translations in a hybrid block mapping mode, or to increase the length of sequentially stored logical pages. Our evaluation examines, for the first time, the relationship between scrubbing frequency and its effect on write amplification when used to reduce the memory footprint for address translation.

The **third contribution** is the design of an FTL for SSDs that internally compress data. Unlike related work using a bitmap-based approach [9], our approach can ensure a compact representation of address translation, and unlike learned-index-based FTLs [33], it also keeps the memory footprint small even at high and non-uniform compression ratios.

We implemented HyTorC and competing approaches inside an Open Channel SSD to realistically investigate the impact of our new approach on memory consumption, performance, and write

amplification. We evaluated the address translation techniques using Microsoft Cambridge traces (MSR) [24] and Alibaba Cloud block traces [20, 35]. The MSR traces were collected from 36 volumes of an enterprise data center over the duration of one week and include access patterns of enterprise services like file servers, web servers, and web caches. The more recent Alibaba block level traces were collected from 1,000 randomly sampled virtual disks from Alibaba Cloud production systems over the duration of one month. The MSR and Alibaba traces are among the largest real-world block-level traces available to researchers and academics.

The evaluation shows that HyTorC requires, for the Alibaba Cloud block traces, on average 98.9% less memory than page mapping, 95.6% less memory than the caching-based DFTL [5], and 87.3% less memory than S-FTL [9]. A surprising result is that HyTorC also reduces the memory consumption compared to LeaFTL [33] by 56.5%, showing that compact, hand-crafted data structures can outperform learned indexes. Similar results also hold for the MSR traces.

The resulting memory savings can be used to improve performance by providing more memory for caching. However, we expect SSD manufacturers to optimize SSDs for Cloud and archive workloads for cost, so that they will simply provide less memory inside the SSD controllers. The investigation on whether using saved memory for caching can improve performance is therefore beyond the scope of this article.

One limitation of the Open Channel SSD implementation is that the underlying dual-core CPU is weak even compared to today's SSD controllers, so it was not possible to hide all of the performance impact of scrubbing by offloading scrubbing to background jobs on additional CPU cores. Also, the Open Channel SSD does not include a compression unit, so we were unable to compress data on the SSD itself. We discuss these restrictions in the evaluation section.

The remainder of this work is structured as follows. Section 2 describes the related work on address translation schemes, while Section 3 describes the new hybrid address translation algorithm. Section 4 presents the evaluation results and we will conclude in Section 5.

2 Model and Related Work

This section gives a short introduction into the applied SSD model and will then present related work on compression inside SSDs and address translation.

2.1 SSD Model

The capacity of SSDs is partitioned into n blocks and each block is able to store m physical pages. A physical page is the smallest granularity, which can be individually addressed and it has a fixed capacity c_p. The individual cells inside a page can only be written in one direction before being reset by an erase operation and the granularity of erase operations is, due to physical constraints, a complete block. Directly overwriting physical pages is, therefore, with the exception when using write-once memory codes [8, 22, 23, 40], not possible. Furthermore, erasing blocks damages the flash cells, so that the number of possible erase cycles per block is limited.

SSDs implement an out-of-place update mechanism to overcome the physical restrictions of flash cells. Incoming write operations are buffered in a small write cache and are then flushed to an open write block. The old versions of the corresponding logical pages are then invalidated. The number of open write blocks is limited to simplify the SSD management.

The **flash translation layer** (**FTL**) manages the mapping between logical and physical pages, constantly reclaims free space by running GC cycles [1, 2], and ensures that the number of erase cycles per block is evenly spread over the SSD. Read and write latencies can increase during GC if these operations are queued behind GC. This effect of GC can be hidden by a concurrent background implementation [11, 16, 38, 41], while orthogonal wear-leveling strategies balance the erase count of blocks [3, 10].

The capacities of SSDs have grown exponentially over the past decades and manufacturers have continually increased the page size c_p and the number of pages per block m. In the following, unless otherwise noted, we use a rather small page size of $c_p = 4,096$ bytes and a number of pages per block $m = 1,024$, which are the device parameters of the open-channel SSD used in our evaluation. All data structures are designed for a maximum physical capacity of 4 PByte, which is 40× larger than the capacities of the largest available SSDs, which can store between 64 and 128 TByte [27, 32, 37]. Scaling beyond 4 PByte requires adjusting the bit width of some variables, which is standard industry practice when moving to the next generation of SSDs.

2.2 Address Translation Techniques

FTLs uniquely map each logical page to a physical page on an SSD. A simple approach, called **page mapping**, stores one table entry for each logical page [39]. This approach is the fastest mapping approach because each mapping involves only a single table lookup and the GC strategy can map each logical page to any physical page, but at the cost of high memory consumption.

The other extreme is to partition logical pages into logical blocks of the same size as physical blocks, storing only one entry for each logical block, and ensuring that each block contains only sequential pages. This **block mapping** results in write amplification for non-strictly sequential workloads that can be proportional to the number of pages in a block [14].

There have been many approaches to strike a balance between page mapping and block mapping [15]. **Hybrid mapping schemes** try to map most pages based on a block mapping scheme. In addition, they introduce log blocks with fewer mapping restrictions than block mapping. Implementations of hybrid mapping schemes range from storing only pages from a single block in a log block to supporting full page mapping within log blocks [12, 17, 19].

Many I/O workloads have temporal localities and hot address translations can therefore be **cached in memory** while most mappings are stored on the SSD. DFTL, therefore, distinguishes between data pages and translation pages and stores both on the SSD [5]. Each translation page contains the page mapping for a contiguous list of logical pages. DFTL caches the mappings for the most recently used logical pages and a timestamp used by the cache eviction strategy. It also requires an in-memory **Global Translation Directory** (**GTD**) to locate translation pages on the SSD. The I/O overhead of DFTL can be limited to at most two additional page reads and one write per cache miss.

Several approaches have improved on the original DFTL approach. HAT (**h**iding the **a**ddress **t**ranslation) provides an additional small non-volatile cache to store the page table and hide lookup latencies [7]. S-FTL compacts the in-memory representation of translation pages by storing only a single entry for sequentially stored logical pages and introducing a bitmap to detect these sequences [9]. CDFTL introduces a second cache layer that stores mappings evicted from the first cache layer to delay and batch write-back operations [29]. TPFTL introduces a two-level LRU strategy that uses the hotness of logical pages and complete translation pages. It also detects sequential accesses that pollute the cache and batch-updates translation pages [44].

HyTorC builds on S-FTL's idea of compacting the representation of sequentially stored pages using bitmaps. HyTorC further reduces memory consumption by adding a block mapping mode, compressing bitmaps using run-length encoding, and using a scrubbing process to reassemble randomly written pages. In addition, HyTorC supports data compression and does not use caching to further minimize the memory footprint, but always keeps all page mappings in memory. Caching is orthogonal to HyTorC and can be used in cases where even its extremely high efficiency is not sufficient.

LeaFTL uses learned indexes to learn the logical to physical page mapping within a block using linear functions [33]. It reduces memory consumption of mapping tables by encoding contiguous

pages in a single linear function. Further memory reductions are possible by allowing extra I/Os to overcome bounded mispredictions. LeaFTL also benefits from resorting pages before writing them within a block according to their logical block number. LearnedFTL [36] additionally combines the learned indexes with the demand-based FTL scheme TPFTL [44].

Similar to learned indexes, HyTorC benefits from sequential mappings using its run-length encoding. HyTorC additionally reduces the memory footprint by performing a background scrubbing, and it directly supports logical page compression. We will show that lessons learned from both conventional data structures and learned indexes can help to design even more efficient page mapping approaches.

2.3 SSD Compression

Compressing data can reduce write amplification and improve SSD performance. Zuck et al. examined the tradeoffs of whether compression should be implemented in the application, the storage stack, or the SSD itself and concluded that compression inside the SSD is the superior choice [45].

Most early compression-enabled SSDs uniquely map each logical page to a physical page, while a physical page can store more than a single logical page [18, 26]. This means that two or more compressed logical pages are stored in their compressed form in a physical page only if they fit together in that physical page. This reduces compressibility compared to being able to spread logical pages over multiple physical pages [45], but also simplifies the FTL design. BlueZIP SSD, for example, then simply uses a page mapping approach to find address translations, but needs an additional data chunk table for GC to distinguish between physical pages that store a single or multiple logical pages [18].

IBU (In-Block Update) is a hybrid address mapping scheme that maps sequential, non-compressible data as blocks and uses page mapping to map the remaining pages [34]. The evaluation of IBU is based on simulations only and shows that IBU reduces the size of mapping tables by a rather small factor of 4× compared to page mapping.

Commercial SSDs that support compression either use the extra capacity to increase over-provisioning and therefore SSD endurance [6] or to provide more virtual capacity [28]. These features can be used transparently or supported by new data structure designs [4, 28].

It is not publicly known whether commercial SSDs supporting compression distribute a logical page across multiple physical pages. In the following, we assume that logical pages are stored in their compressed form only if at least two compressed logical pages fit into one physical page. However, all of our proposed data structures can be adapted to the case where a logical page is spread over two physical pages.

3 Design

This section first presents a set of requirements for address translation algorithms inside SSDs and then presents a new approach to significantly decrease the memory consumption of address translation while supporting the requirements of new scale-up flash devices.

3.1 New Requirements for Address Translation

The increasing capacity of SSDs combined with support for compression requires a re-design of the address translation unit inside the FTL of an SSD:

— First, it is necessary to significantly reduce the memory footprint of the address translation unit compared to page mapping to keep the cost of main memory within the SSD low. This implies that it is too costly to maintain a pointer to a physical page for each logical page, and therefore, the address translation unit should group pages so that a group of pages can use a single pointer [9, 33].

— Second, new SSDs compress data to increase the storage capacity provided. This means that the current state of bitmap-based compaction cannot be used in conjunction with data compression [9] because compression breaks the nice sequential patterns they use and results in significantly increased memory consumption.
— Third, the read performance of an SSD is limited by the latency of accessing a physical page, and the SSD must support random access patterns. Therefore, I/O accesses required by the address translation unit should be minimized as they can double the cost of random reads, and the compression of the translation table must be high enough to avoid the need for caching schemes such as DFTL.

HyTorC combines and introduces several techniques to fulfill these requirements. HyTorC follows a hybrid design that keeps as many address translations in block mapping mode as possible. For page mapped address translations, it detects, similar to S-FTL, sequential address translations and encodes them by a single pointer for each sequential sequence and one bitmap to detect the end of sequences (Section 3.2). The idea of using bitmaps is extended by an additional bitmap to encode data compression and by aggressively compressing metadata (Section 3.3). Scattered cold data leading to many address translations stored in page mapping mode is re-grouped using background scrubbing (Section 3.4).

We describe the address translation for reads and writes in Section 3.5. The on-disk layout of our address translation (Section 3.6), the GC strategy (Section 3.7), and the crash recovery (Section 3.8) were implemented very close to standard page mapping strategies.

3.2 Logical Block Descriptor (LBD)

HyTorC is based on the concept of logical blocks, where the number of pages assigned to each logical block is equal to the number of uncompressed physical pages m that can be stored in a physical hardware block.

We provide an LBD for each logical block (see, e.g., Figure 2). LBDs can be of different sizes, include different fields, and the required memory for LBDs is allocated on demand. HyTorC indexes LBDs using an array with one pointer for each LBD. HyTorC maps an LPN to its corresponding logical block number within the array by taking the modulo of the LPN by the number of physical pages m in a block. The LBD is then used to derive the physical location of the corresponding logical pages.

We will introduce the additional fields of the LBD using an example in the following paragraphs, which first describes the block mapping mode and then continuously evolves.

Block mapping: We will start the description of the LBD for a logical block which is completely stored in block mapping mode. Each LBD begins with the logical block number corresponding to the LBD, followed by four bit flags and a 28-bit timestamp.

The unpredictability of incoming writes can lead to a situation where HyTorC can store a complete logical block at once, but cannot start writing it at the beginning of a physical block. A logical block can, therefore, start at any position within a physical block. The example in Figure 2 shows such a situation where the first logical block with LBN 0 was stored as a single entity in block mapping mode. However, LPNs 0–1 were assigned to the end of physical block 2, while the remaining pages were assigned to the beginning of physical block 0. Therefore, HyTorC provides two block pointers. Each block pointer has a size of 40 bits to map a block to an arbitrary PPN, resulting in an address space of 4 PByte for 4 kByte pages. The maximum number of block pointers is limited to two, because the FTL can always start writing the second part of a logical block at the beginning of a physical block during GC or scrubbing.

Correctly locating a page within a block requires taking page compression into account. HyTorC therefore stores an additional bit for each logical page in the compression map $cm0$ of the block

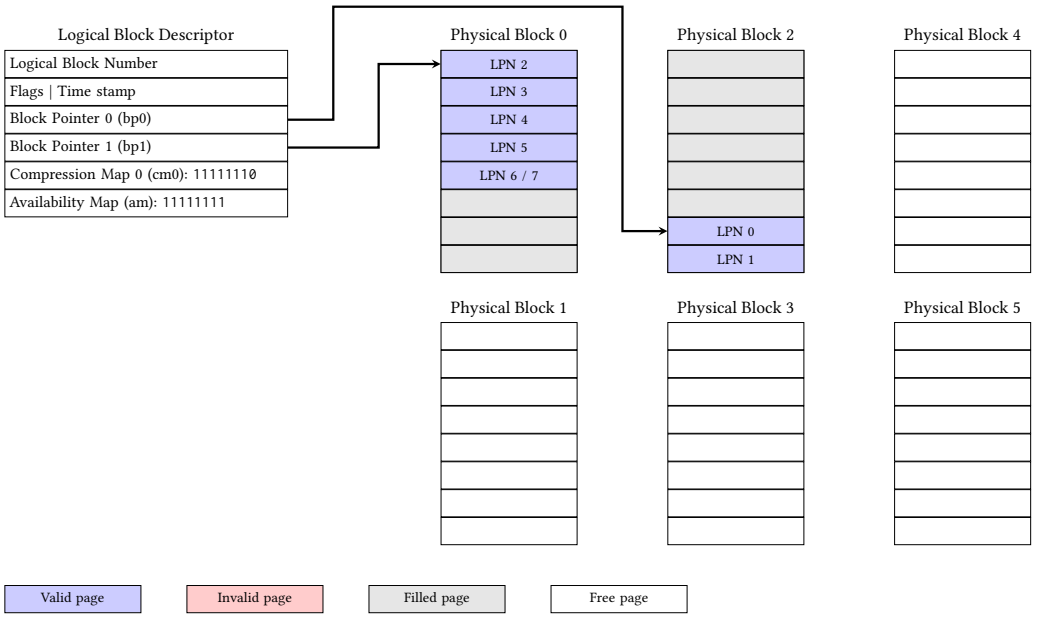


Fig. 2. Mapping between LBD and physical SSD pages. In the example, each logical and physical block consists of eight pages and the SSD consists of six physical blocks. If two or more logical pages are compressed into a single physical page, both their indices are added to the description. E.g., LPN 6/7 means that the logical pages 6 and 7 are compressed into one physical page.

descriptor, that indicates whether this page starts at a new page boundary or is co-located with the previous page. A bit in the compression map is set to 1 if the corresponding logical page starts at a new physical page. In our example, the eights bit of the compression map is set to 0 and therefore indicates that pages 6 and 7 of the logical block are compressed and mapped to the same physical page.

The compression map defines the state when the block mapped pages have been written and it also includes information about logical pages, which later have been transformed to page mapping mode. When locating the offset of a logical page in block mapping mode, it is therefore possible to simply count the trailing bits in this compression map.

The LBD includes an availability map *am* that indicates for each logical page whether it is part of the block mapping or whether it is mapped individually in page mapping mode. A bit in the availability map is set to 1 if the page is stored in block mapping mode. In the example, all bits are set to 1, indicating that all pages are still stored in block mapping mode.

Page Mapping: Figure 3 extends our example and shows the situation after first LPN 0 was overwritten and stored inside physical block 3, and then LPN 2, LPN 3, and LPN 5 were overwritten and stored inside physical block 1. For the four pages involved, the corresponding bits in the availability map have been set to 0, so that HyTorC can distinguish between pages stored in block mapping mode and page mapping mode.

HyTorC always sorts logical pages by their LPN before writing them to a physical block. Accordingly, LPN 2 and LPN 3 were stored sequentially, so it is sufficient to store a single pointer for them. Therefore, we introduce an additional m bit sequentiality map *sm*, where a bit is set to 1 if the page is not stored sequentially and, therefore, requires its own page pointer. In our example, the bits inside *sm* for LPN 0 and LPN 1 are set to 1, the bit for LPN 3 is set to 0. LPN 5 directly follows LPN 3

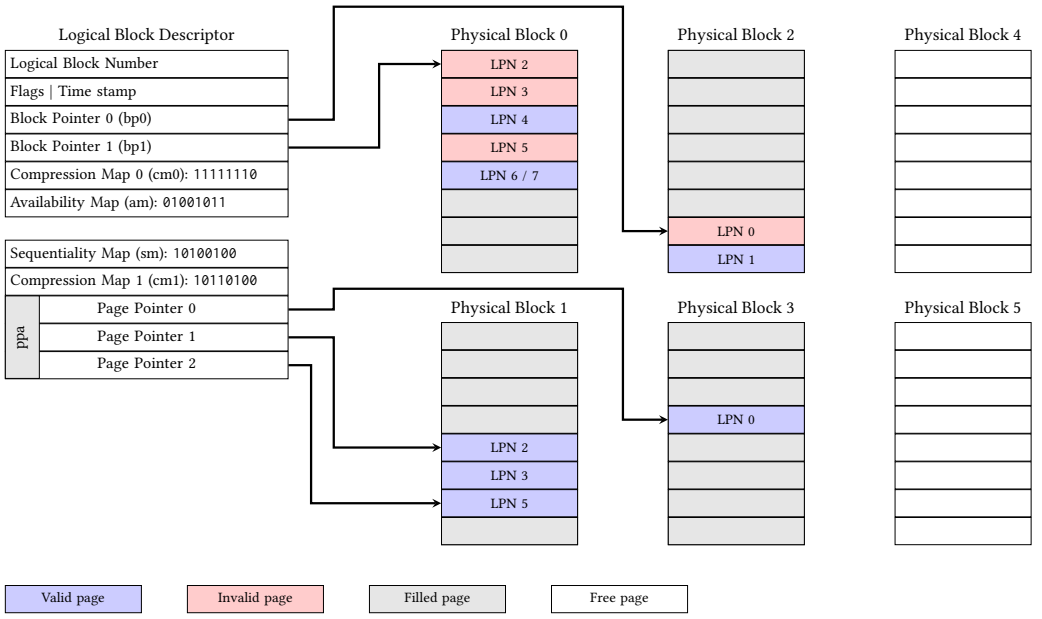


Fig. 3. LBD after several pages of logical block 0 have been written in page mapping mode.

within block 1, but still requires an additional page pointer. The reason is that HyTorC stores only one *sm* bitmap for all page-mapped logical pages within an LBD, and therefore, we can construct situations where it is impossible to derive the correct location for a logical page without such an additional pointer.

In addition, sequentially written pages can also be compressed. Therefore, we introduce a second m bit compression map $cm1$, which again stores a 1 if the sequentially written logical page in page mapping mode is stored in a new page after its logical predecessor. These compression bits may differ from the compression bits in $cm0$ because the compressibility of a logical page may change when it is overwritten. Finally, the page pointers are stored in the page pointer array ppa at the end of the LBD.

3.3 Size of the LBD and Compressing the LBD

The minimum size required to store the LBD depends on several parameters. The header, therefore, contains four status bits, where the first bit indicates whether the second block pointer is needed (saving 40 bits), and the other three bits indicate whether an availability map *am* is necessary (only if not completely block mapped), whether the compression maps $cm0$ and $cm1$ are necessary (only if compression is possible within this block), and whether the page mapping part contains a sequentiality map *sm*.

The m-bit wide maps can be compressed in most cases. Therefore, we start each map description with a 5-bit length field, indicating the number of 32-bit values for the map. The maps are stored as either plain bitmaps or run-length encoded. We use an extra three bits to encode the number of bits used per run-length entry, so that up to 256 bits can be encoded by a single entry. We set the extra three bits to zero if the bitmap is stored as a plain bitmap.

If each page is stored in block mapping mode and only one block pointer is needed, the minimum size for an LBD is 104 bits. In this case, the LBD is padded to 128 bits up to the next 32-bit limit. Comparing this to a page mapping approach, where each entry within a page mapping requires 40

ALGORITHM 1: Translating logical to physical page addresses

Require: Logical page number *addr* and LBD

$p = addr \bmod m$
if $(am[p] == 1)$ **then**
  {Page is block mapped}
  po = leadingOnes($cm0$, p)
  $bo = bp1 \bmod m$
  if $(bo + po > m)$ **then**
    return $bp2 + bo + po - m$
  else
    return $bp1 + po$
  end if
else
  {Page is page mapped}
  o_{pma} = leadingOnes(sm, p) - 1
  p_{base} = indexOf1Bit(sm, o_{pma})
  po = leadingOnes($cm1$, p) - leadingOnes($cm1$, p_{base})
  return $pma[o_{pma}] + po$
end if

bits, the compression ratio of the LBD compared to page mapping can be as high as 320× in the best case, and we will show that the average compression is as high as 92× in our evaluation.

3.4 The Scrubbing Process

Updating pages and continuously running the GC process will cause most pages to be mapped in page-mapping mode over time, and the memory savings of the LBD will disappear. Therefore, a scrubbing process is constantly running in the background, waking up f_{scr} times per second to merge one logical block in each iteration. It iterates through the list of LBDs and selects as the next scrubbing block an LBD that has not been written to for more than t_{scr} seconds and that stores more than p_{scr} pages in page mode.

Blocks used for scrubbing are outside the standard write process, so that recent writes are not mixed with these cold blocks. The values for f_{scr}, t_{scr}, and p_{scr} allow to control the bandwidth required by scrubbing and, therefore, to adequately use the additional internal read and write bandwidth, which today's SSDs can deliver beyond the required external bandwidth. The default values in our tests, if not specified otherwise, are $f_{scr} = 10$, $t_{scr} = 300$, $p_{scr} = 512$.

3.5 Reading and Writing Pages

This section describes the process of using and updating the LBD for reading and writing pages. We therefore present the corresponding algorithms and use Figures 2 and 3 as examples to explain the process.

Reading a logical page: Reading a page requires a translation between its logical and its PPN. The inputs of the corresponding algorithm are the LPN *addr* and its LBD (see Algorithm 1). In a first step, the algorithm calculates the logical offset p of the page within the logical block and checks within the availability map *am* whether the page is stored in block mapping mode or page mapping mode.

An example of reading a logical page stored in block mapping mode is shown for LPN 4 in Figure 3. Its corresponding bit in the *am* bit map is set to 1, indicating that the page is stored in block

mapping mode. The calculation of the physical offset of LPN 4 in block mapping mode must include the number of compressed pages stored before LPN 4. This physical offset po can be calculated by simply counting the number of 1 bits stored before the p'th bit of the compression map $cm0$ (implemented in the function `leadingOnes()`), which results in four uncompressed pages stored before LPN 4.

The logical block can be split into two physical blocks. In Algorithm 1, we therefore, first compute the offset bo of the first block pointer $bp0$ into its corresponding physical block. In the example, bo computes to 6, and the `if` clause in Line 6 detects that $bo + po$ is bigger than the number of pages in a block and follows the second block pointer $bp1$ to derive the location of LPN 4.

Exactly the same logic is applied when reading a compressed page. In our example, calculating $p0$ would result in 6 for LPN 7, so the following read function would load the page that stores both LPN 6 and LPN 7. We use the **out-of-band** (**OOB**) area of each physical page to store information about whether more than one logical page is stored within it, and where the boundaries between these logical pages are within the physical page.

Calculating the PPN of a logical page stored in page mapping mode is slightly more complicated. The LBD stores a pointer for each set of sequentially stored logical pages inside the page pointer array ppa. The offset o_{pma} into the ppa for the starting page p_{base} of the sequentially stored set of pages, which includes the sought page, can be calculated by counting the number of leading 1-bits in the sequentiality map sm before position p and by then subtracting 1. In case of LPN 3 in Figure 3, this leads to an offset of 1 and we can use in the following page pointer 1.

To derive the logical distance between the sought logical page p (LPN 3 in the example) and the starting page p_{base} of the first page of its sequential run (LPN 2 in our case), it is necessary to receive the address p_{base} by deriving the index of the o_{pma}-th 1-bit inside sm using the function `indexOf1Bit()`. Without compression, the physical address of the sought page can now be calculated as $pma[o_{pma}] + (p - p_{base})$.

Including compression, we also have to consider the number of compressed pages between page p and p_{base}, which can be calculated based on the second compression map cm_1. We have to subtract the number of new uncompressed starting positions up to p_{base} from the number of uncompressed starting addresses from p. These can again be derived using the `leadingOnes()` function. The further handling of read requests after the address translation follows the standard read process of an SSD.

Writing a set of logical pages: Algorithm 2 illustrates the process of updating the LBD after sequentially writing a set of n pages. The algorithm first updates in Lines 1–4 the availability map am to 0 for each page, because the pages are later stored in page mapping mode. This operation does not produce a negative side-effect even if the page has been stored in page mapping mode before.

Lines 5–12 set the physical start address for the first page of the sequential write. In the case that the page has not yet an entry in the page pointer array ppa, a new entry is created at the offset po inside the array. This offset can be computed based on the number of logical pages inside the logical block, which are also stored in page mapping mode, which have a smaller LBA than this write, and which are also the start page of a sequential write by calling `leading_ones(sm, p)`. If there is already an entry in the ppa for this page, then this slot at position po is reused. Afterward the bits for the sequentiality map sm and the compression map $cm1$ are set to 1, indicating that the page starts at a new position and is not compressed together with its predecessor.

Lines 13–24 update the remaining pages of the sequential write. If some of these pages have been the start of a previous run of sequential pages, their entries must be removed from the ppa and in all cases, we store the previous physical position of the last handled page in $paddr\_old$ for later housekeeping. Then, we unset $sm(p)$ for these pages and update their compression bits in $cm1$.

ALGORITHM 2: Updating LBDs after sequential page writes

Require: Logical start page *l_addr*, number of sequentially written pages *n*, new physical page numbers *p_addr*[*n*], old physical start address *paddr_old*, and LBD

```
p = l_addr mod m
{Set all pages of sequential write to page mapping mode}
for (i = 0, i < n; i++) do
    am[p + i] = 0
end for
{Set the bits and physical start address for first page}
po = leadingOnes(sm, p)
if (sm[p] == 0) then
    ppa.insert(po, p_addr)
else
    ppa[po] = p_addr
end if
sm[p] = 1
cm[p] = 1
{Update remaining pages}
for i = 1, i < n; i++ do
    if (sm[p + i] == 1) then
        paddr_old = ppa[p0 + 1]
        ppa.delete(p0 + 1)
    else
        paddr_old = paddr_old + cm1[p + i]
    end if
    sm[p + i] = 0
    if (p_addr[i - 1] == p_addr[i]) then
        cm1[p + i] = 0
    else
        cm1[p + i] = 1
    end if
end for
{Clean-up}
if ((p + i + 1) < m - 1) && (sm[p + i + 1] == 0)) then
    ppa.insert(po + 1, paddr_old + cm1[p + i + 1])
    sm[p + i + 1] == 1
    cm1[p + i + 1] == 1
end if
```

Additional care must be taken if a page is the last page of a write and the next page was part of a previous sequential write. Therefore, if $sm(p+1) = 0$, the current physical address of the next page must be computed by taking the start address of the previous sequential write and adding the physical distance to the next page. Then we can insert it at position $po + 1$ into the physical address mapping table. Afterward, we have to set $sm(p + i + 1) = 1$ and $cm1(p + i + 1) = 1$.

Finally, the time stamp inside the LBD is updated after the write process to the current time *t*.

Use a write operation to LPN 5 in Figure 2 as an example. The pages LPN0, LPN2, and LPN3 have already been written before. The mode of LPN 5 is transferred to page mapping and the bit at

Timestamp (28 bits): Time when the physical page was written
Status Flags: ECC status, bad block marker, etc. (hardware-dependent)
Logical Page Count (4 bits): Number of logical pages stored inside this physical page, e.g., 3
LPN Map (Logical Page Count x 32 bits): Mapping of ppn to lpn, e.g., `LPN[0]`, `LPN[1]`, `LPN[2]`
offset (Logical Page Count x 12 bits): offset of logical pages, e.g., `Offset[0]`, `Offset[1]`, `Offset[2]`

Fig. 4. OOB area of a physical page.

its offset p in the availability map *am* has to be unset to 0. If the page is the first page of a write request, which is the case for LPN 5, we have to set its bit in *sm* and have to insert its physical address at position `leadingOnes(sm, p)` into the page mapping array *pma*. If the page is part of a larger write request and it sequentially follows its predecessor, then we have to set its compression bit in *cm*1, iff it is not stored in one physical page with its predecessor logical page. The resulting LBD after writing LPN 5 is shown in Figure 3.

3.6 Physical Block Descriptor

The OOB area of each physical page stores the time the page was written and how many logical pages are stored in that physical page (see Figure 4). If multiple compressed logical pages are mapped to a physical page, it also includes a mapping from LPNs to physical addresses inside that page for each of the logical pages stored in it.

The OOB area is used to recover persistently stored information after a catastrophic failure (see Section 3.8), and it is additionally required during GC (see Section 3.7). The OOB area is also accessed by the FTL during reads, so that the FTL can uniquely identify the start position of each logical page within the physical page.

3.7 GC

GC ensures that there are always enough free blocks to persist new page writes. GC can be in the critical path of write requests and it is important that it can find and clean a victim block as fast as possible. We, therefore, decided against coupling GC with a process that optimizes the sizes of the LBDs. GC is triggered in our implementation when the number of free physical pages falls below a threshold of 10%.

HyTorC can use any standard GC algorithm and we restrict the discussion inside this article to the wide-spread Greedy algorithm. Greedy selects the block with the minimum number of valid pages as its next victim. Greedy achieves an optimal write amplification for random access patterns [1, 2], whereas its write amplification can significantly increase for real-world access distributions.

SSD compression makes it more complicated to decide which block contains the minimum number of valid pages, as a huge number of logical pages can be compressed into a very small number of physical pages [18]. HyTorC approximates the fill level of a block by keeping the two counters `maxpgs` and `curpgs` per block. Both counters are initialized with the number of logical pages initially written to a block. `maxpgs` keeps this maximum number of logical pages, whereas the current number of logical pages `curpgs` is decremented every time when a logical page in the block is invalidated. Greedy then selects the block with the smallest value $\left\lceil m \cdot \frac{curpgs}{maxpgs} \right\rceil$. The

normalization with the number of physical pages per block m and the ceiling-function is chosen to allow for $O(1)$ victim selection using m lists to sort blocks according to their number of valid pages [31].

After selecting the next victim block, and if $curpgs > 0$, GC loads all physical pages into memory, checks whether they contain valid pages according to the OOB area and the corresponding LBD, and writes valid pages back to the next free write block using the standard write approach. It then updates the LBDs of the logical blocks that are part of the physical address mapping and deletes the victim block.

3.8 Crash Recovery

This article adopts the crash recovery method for flash storage devices proposed in [42]. The strategy periodically creates synchronous checkpoints that preserve critical information about the state of the storage device and persistently stores this information on the SSD. Checkpoints include logical-to-physical page mappings, block status, and page status, among others. The algorithm uses the information stored in checkpoints and OOB areas to quickly and accurately restore the device to a stable state.

Recovery starts by reloading the last checkpoint. It then checks the first page of each block to see if it has been written since the last checkpoint, and sorts blocks written after the last checkpoint according to their timestamps. Recovery then recreates the insertion order of the page writes into the mapping structure.

Reading only one of 1,024 pages per block for blocks written before the last checkpoint reduces the recovery effort by nearly 1,024 times compared to a naive approach. Storing checkpoints at the default frequency of once per minute then ensures that all necessary OOB areas of newly written pages can be read in less than one second.

4 Evaluation

This section presents the evaluation results for HyTorC and compares it to a **page-mapped FTL** (**PFTL**), DFTL, S-FTL, and LeaFTL. All experiments were performed on an open-channel SSD, and we used Alibaba Cloud traces [20] and MSR Cambridge (MSR) traces [24] to provide a realistic comparison of our memory consumption, read and write performance, and write amplification for real-world applications. The goal of this section is to show that HyTorC provides much better memory efficiency than competing approaches, with similar performance to PFTL. In addition, we expect that HyTorC, if properly tuned, may also have a positive effect on wear-leveling, but leave these investigations to future work.

4.1 Experimental Systems

This subsection introduces our hardware platform, describes the subset we selected from the Alibaba Cloud traces and the MSR Cambridge traces, and also briefly describes the different FTLs we compare.

4.1.1 Hardware Platform. We evaluated our FTL design based on implementations on an FPGA-based open-channel SSD platform that is equipped with a Xilinx Zynq XCZU7CG-FFVC1156 programmable SoC. This SoC embeds a dual-core ARM Cortex-A53 processor running at 1.2 GHz, along with a dual-core Cortex-R5 processor running at 500 MHz. The ARM side (PS) and the FPGA side (PL) each have 4 GB of memory. The SSD platform communicates with the host via a PCIe 3.0 × 8 interface, providing up to 8 GB/s bandwidth. The SSD platform uses eight 128 GB flash memory chips, the page size is 4,096 bytes, the number of pages m per block is $1,024$, and we selected an

Table 1. Alibaba Trace Details

Id	Capacity in GByte	Million Reads	Million Writes	I/O in GByte	Avg. Req. in kByte	WSS Ratio
7	537	12	253	2,441	9.2	5.8%
10	86	2,754	3	12,530	4.5	39.4%
743	429	2	1	205	70.8	26.0%
801	537	≈ 0	2	830	380.7	93.7%
804	1,099	248	48	33,607	113.5	96.1%

overprovisioning rate of 20% and a GC threshold of 10%. The flash write latency is $1,200\mu s$ and the read latency $220\mu s$.

All tested FTLs are fully implemented within the open-channel SSD platform. The embedded A53 processor runs the main part of the FTL, processes I/O requests from the host and sends flash data I/O commands to the R5 processor, while the embedded R5 processor interacts with the ONFI flash controller to manage flash data read and write operations.

The open-channel SSD is integrated into a server with an Intel Core i9-10900X processor running at 3.7 GHz with 32 GB of RAM. This host side is only responsible for reading traces and sending I/O requests to the open-channel SSD.

4.1.2 FTL Implementations. We implemented two versions of our HyTorC FTL and compared them with a standard PFTL, the DFTL [5], the S-FTL [9], and the learned index-based LeaFTL [33]. All FTLs had to be implemented by us, since no source codes were available, respectively, the source code for LeaFTL is only available as Python code, which cannot be used for our Open Channel SSD. We followed the implementation descriptions in their articles and used their default parameter settings. For all FTLs except DFTL, we allocated enough memory to store the complete mapping in main memory. For DFTL, we used **Least Recently Used** (**LRU**) to evict mappings and set the maximum cache size to cover 10% of the virtual addresses. For LeaFTL, we set the approximation error to its default of $\gamma = 0$, so that each page could be read by a single I/O. HyTorC with run-length encoding implements the full HyTorC stack, while HyTorC$_{NoC}$ without run-length encoding does not compress the bit arrays within the mapping table. Comparing HyTorC and HyTorC$_{NoC}$ therefore directly shows the effect of compressing the bitmaps inside the LBDs.

All FTLs use the same greedy GC and crash recovery approach. Checkpoints of the logical-to-physical mapping are written once per second. In addition, the HyTorC FTLs also run the scrubbing process at the default scrubbing frequency of $f_{scr} = 10$ unless otherwise noted. We have not included existing FTLs for compression-based SSDs because their FTL designs are either not public or, in the case of IBU, only achieve efficiencies similar to D-FTL [34].

4.1.3 Traces and Workload Generation. The first dataset in our experiments originate from a collection of block-level storage traces provided by the Alibaba Group [20, 35]. This trace data includes various operations on storage devices, including reads, writes, and associated metadata information.

Due to the significant size of these traces, we extracted a subset of five traces from the full list to test our FTL implementations. The first criteria was that all traces had to perform a considerable amount of I/O, starting at 200 Gbytes and going up to several Tbytes. We also wanted traces that were read-heavy or write-heavy, as well as traces with a mix of reads and writes. The final criterion was that we wanted to evaluate traces with different average I/O sizes. Table 1 shows the Alibaba Trace ID, the logical size of the corresponding volume, the number of read and write I/O commands,

Table 2. MSR Trace Details

Id	Thousand Reads	Thousand Writes	I/O in GByte	Avg. Req. in kByte	WSS Ratio
Hm_0	1,418	2,576	30.4	8.0	1.4%
Hm_1	581	28	8.8	15.2	2.5%
Mds_0	144	1,067	10.6	9.2	3.3%
Mds_1	1,521	117	88.7	56.8	46.4%
Prn_0	602	4,983	59.1	11.1	6.5%
Prn_1	8,464	2,770	212.1	19.8	37.6%
Proj_0	527	3,697	153.2	38.0	1.6%
Proj_1	21,143	2,497	775.9	34.4	80.1%
Proj_2	25,642	3,625	1,184.6	42.4	79.7%
Proj_3	2,128	116	20.9	9.7	21.5%
Proj_4	6,370	96	144.6	23.5	21.5%
Prxy_0	384	12,135	56.8	4.8	2.0%
Rsrch_0	134	1,300	12.2	8.9	1.7%
Rsrch_1	0	14	0.2	12.2	17.4%
Rsrch_2	136	71	0.8	4.1	8.0%
Src1_0	21,113	16,303	1,538.3	43.1	26.7%
Src1_1	43,576	2,170	1,516.0	34.7	26.7%
Src1_2	484	1,424	53.0	29.1	0.8%
Src2_0	177	1,381	10.7	7.2	1.5%
Src2_1	644	14	37.2	59.3	16.6%
Src2_2	351	806	62.1	56.3	16.6%
Stg_0	308	1,722	22.4	11.6	1.1%
Stg_1	1,400	796	85.5	40.8	9.9%
Ts_0	317	1,485	15.5	9.0	2.1%
Usr_0	904	1,333	48.4	22.7	1.6%
Usr_1	41,426	3,858	2,135.4	49.4	80.1%

and the average I/O sizes for the different traces. In addition, the **working set size** (**WSS**) ratio shows the percentage of logical addresses accessed by this trace.

In addition to the Alibaba traces, we also compare the memory consumption and performance for the different FTLs for the MSR traces (see Table 2). The MSR traces do not contain information about the capacity of the backend virtual volumes, so that we set this, like in previous work, to 1 TByte. Furthermore, these traces are in general slightly smaller.

For each test, we limited the physical capacity of the SSD to 1.2× the size of the corresponding volume, resulting in an overprovisioning of 20%. Before each experiment, we first sequentially wrote all the logical pages of the SSD once, and then used 20% of the trace to warm up the system, so that the system was not perfectly balanced. We then used the remaining 80% of the trace data to run the actual experiment.

The traces only contain the access pattern, not the data itself, and we therefore generated a number of pre-computed data blocks and randomly chose which ones to write to flash. For the data compression experiments, we used a normal distribution to compute the expected length of each data block and simply stored the computed number of bytes from the precomputed data blocks. We did not include a compression unit in our software environment because compression within SSDs is typically done in much faster hardware units.

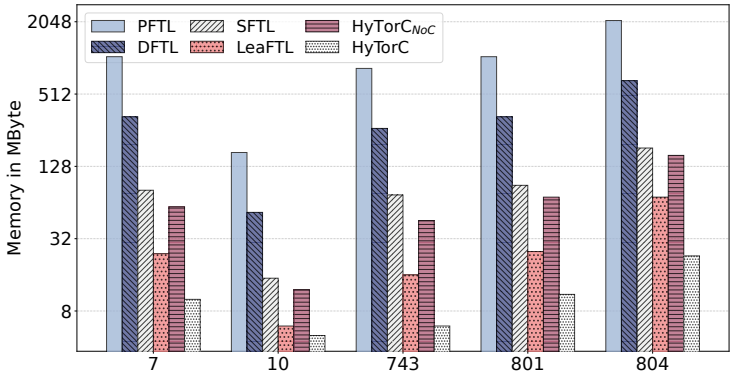


Fig. 5. Memory requirements for Alibaba Cloud traces.

4.2 Memory Usage without Data Compression

In this section, we examine the memory consumption of the FTLs for various workloads without data compression. HyTorC is the only strategy that uses scrubbing and can reduce memory consumption for the mapping tables over time. For a fair comparison, we therefore compare the maximum memory consumption over time, whereas the average memory consumption would have resulted in even greater memory reductions for HyTorC compared to the competing approaches.

The maximum memory consumption over the trace runtimes for each FTL scheme for the selected Alibaba traces is shown in Figure 5 on a logarithmic scale. The memory consumption of PFTL and DFTL depends only on the capacity of the SSD. PFTL must store an entry for each logical page and has the highest memory requirement for all traces. DFTL stores only one entry for each cached page and can theoretically reduce memory consumption by 90% compared to PFTL. However, its cached mapping table stores an additional timestamp for each of its entries and must also store the logical address itself (which is implicitly stored in the PFTL table). Including these overheads, DFTL reduces memory consumption by only 66% compared to PFTL's memory consumption. To achieve a higher compression ratio, DFTL must cache fewer entries.

Figure 5 shows that both implementations of HyTorC including default scrubbing can significantly reduce memory consumption for address translation compared to PFTL and DFTL. HyTorC reduces memory consumption by 97% to 99.3% compared to PFTL (98.9% on average) and by 90% to 97.7% compared to DFTL (96.5% on average). HyTorC$_{NoC}$ reduces memory requirements between 92.3% and 94.7% compared to PFTL (93.5% on average) and between 76.2% and 83.1% compared to DFTL (80% on average). These results also show the importance of using run-length encoding to compress bitmaps. HyTorC is able to reduce memory consumption compared to HyTorC$_{NoC}$ between 58.3% and 86.7% (82.8% on average).

HyTorC$_{NoC}$ and S-FTL share similarities in detecting sequential access patterns. However, HyTorC$_{NoC}$ can reduce memory consumption compared to S-FTL between 16.7% and 37.5% (23.1% on average), because it additionally benefits from encoding parts of the translations in the more efficient block mapping mode and based on the additional scrubbing. Again, HyTorC with run-length encoding is significantly more efficient than S-FTL, reducing memory consumption by 87.3% on average.

Comparing HyTorC with LeaFTL shows that an optimized classical data structure can outperform learned indexes. The memory consumption of HyTorC is between 16.7% and 67.7% (56.5% on average) smaller than the memory consumption of LeaFTL. Again, this can be attributed to the additional benefit of the block mapping part, the run-length encoding, and the scrubbing process.

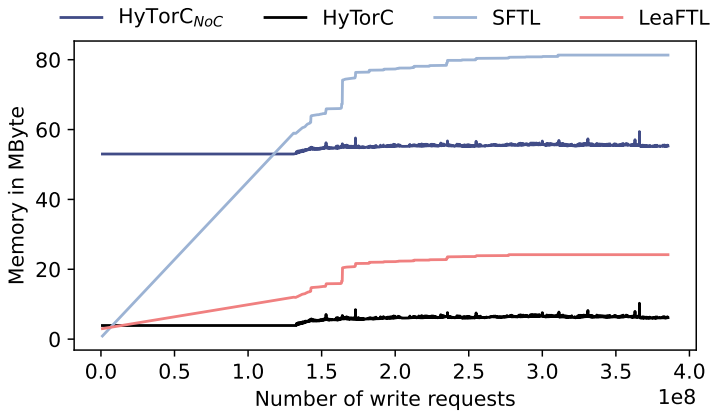


Fig. 6. Memory usage of the FTL schemes for Trace 7.

Figure 6 shows for Trace 7 how the memory consumption evolves over time. We have omitted DFTL and PFTL because their much higher memory consumption would make the differences between the other strategies less visible, and because their memory consumption is based only on cache size/virtual disk size. The runtime is divided into two distinct phases. The first 130 million writes sequentially initialize the SSD, the remaining 253 million writes represent the actual trace.

HyTorC with and without run-length encoding recognizes that it does not need to store the compression bits, which can be indicated by setting the appropriate flag. HyTorC$_{NoC}$ without run-length encoding has a much higher memory consumption than HyTorC. Especially during initialization with sequential writes, HyTorC stores all necessary information with at most two pointers and can compress the bitmaps extremely well. HyTorC$_{NoC}$, on the other hand, cannot compress these bitmaps, even though all their bits are the same (see Figure 2). Each of these bitmaps is 1,024 bits long (number of pages per block) and therefore much larger than the pointer from the logical block address to the physical one.

After the sequential writes were completed and requests based on the trace dataset began to be processed, we observed an increase in memory usage of the page mapping part as random writes progressed. The memory usage stabilized relatively quickly because Trace 7 only accesses a small portion of its volume. Achieving this state of stability depends on the configuration of the scrubbing process (see Section 4.7).

S-FTL and LeaFTL both increase their memory consumption slowly over time and have a similar growth pattern, with LeaFTL's compaction by learned indexes being more effective than the bitmaps in S-FTL. There is a spike in memory consumption for both after the initialization phase ends and accesses become more random, which stabilizes after the main write phase of Trace 7 ends.

The same trends as for the Alibaba traces can also be seen for the MSR traces (see Figure 7). The memory consumption for page mapping and DFTL depends only on the capacity of the backend volume and is therefore constant. S-FTL, on the other hand, is able to detect sequential access patterns and significantly reduce memory consumption compared to page mapping and DFTL. By partially encoding translations in block mapping mode and performing scrubbing, HyTorC$_{NoC}$ further reduces memory consumption between 10% and 40% compared to S-FTL (19% on average).

LeaFTL further reduces memory consumption by an average of 51% compared to HyTorC$_{NoC}$. For the MSR datasets, HyTorC is even able to widen the gap over LeaFTL, so that HyTorC requires on average 61.5% less memory to store its page mapping information than LeaFTL.

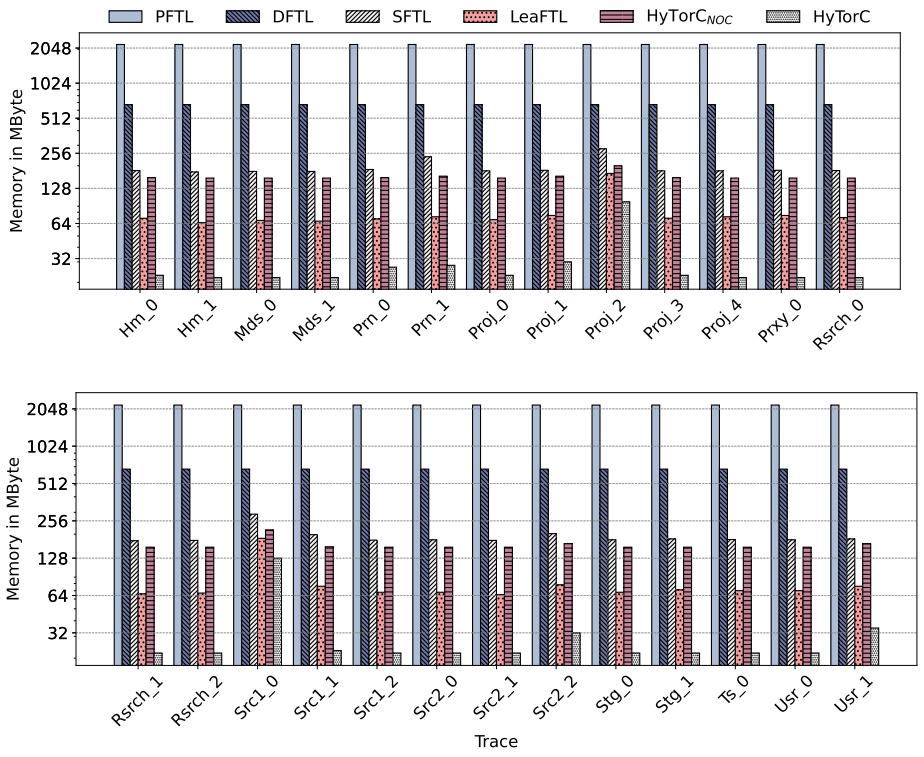


Fig. 7. Memory requirements for MSR traces.

4.3 Performance

This section presents the performance analysis of the proposed FTL solution. The expectation for the performance comparison is that the HyTorC approaches are significantly faster than DFTL, because DFTL requires additional I/Os for caching. Differences compared to PFTL, HyTorC, S-FTL, and LeaFTL can be expected based on the additional computation for address translation and for the scrubbing process. Therefore, we first measured the average computation time for address translation in HyTorC.

The average address translation time for HyTorC$_{NoC}$ is 880 ns with a standard deviation of 45, and HyTorC takes an average of 1,036 ns with a standard deviation of 106, which is significantly less than the average read and write time. The time required for address translation by HyTorC, therefore, has no impact on performance in our evaluation.

Scrubbing is not parallelized in our implementation and, therefore, queues other operations behind it. This effect can be overcome by a more concurrent implementation that hides the scrubbing, as has been shown previously for GC processes [11, 16, 38, 41]. Unfortunately, the CPUs in the Open Channel SSD used in our evaluation did not have enough cores to optimize the overall SSD performance, so we refrained from parallelizing scrubbing. However, we will show in Section 4.7 that the performance of HyTorC and page mapping differs only by the additional computation time when scrubbing is disabled or when it is done in the background.

4.3.1 Write Performance. Figures 8 and 9 show the average write latencies in microseconds of the examined traces. The latency differences between the traces are due to their largely different

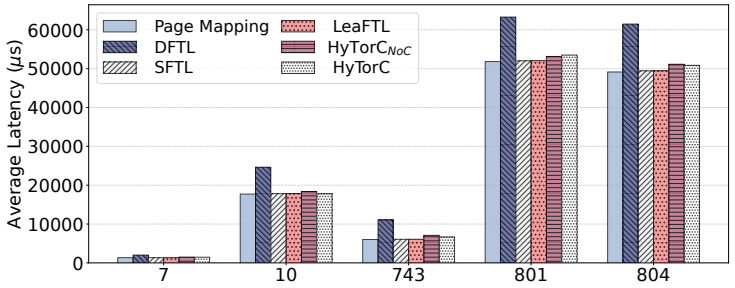


Fig. 8. Average write latencies for Alibaba cloud traces.

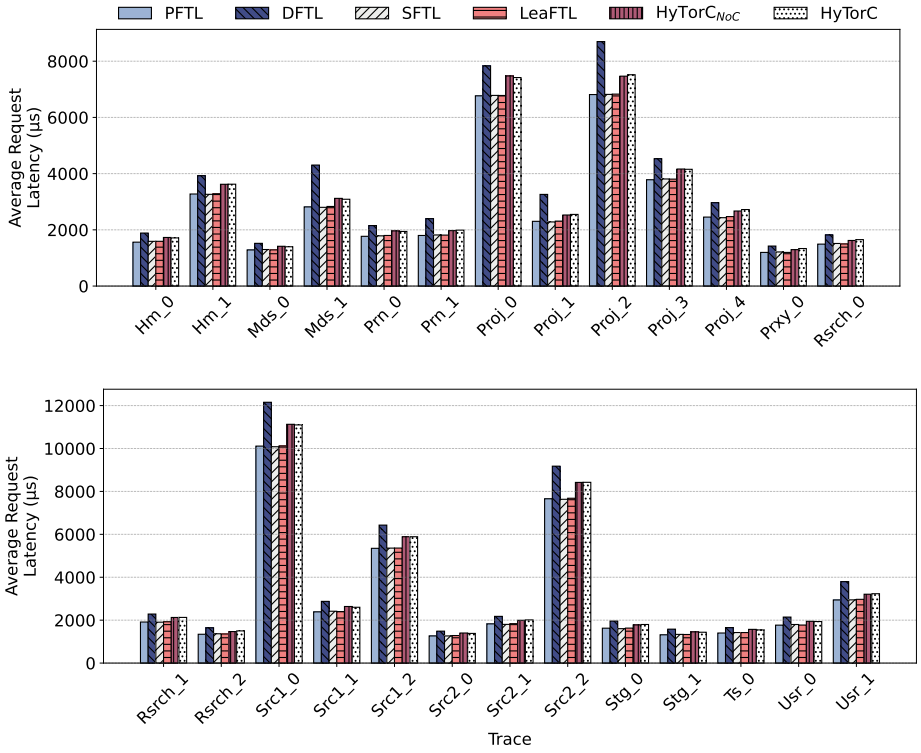


Fig. 9. Average write latencies for MSR traces.

request sizes. As expected, the PFTL is the fastest address translation technique in all cases, closely followed by S-FTL and LeaFTL. The HyTorC approaches have to perform additional scrubbing and are therefore slightly slower.

The relative performance of HyTorC and HyTorC$_{NoC}$ depends only on the bitmap state. If it is relatively simple, HyTorC's array to store the bitmap is shorter and the calculation is slightly faster. On the other hand, if the state in the bitmap is more complex, the performance of HyTorC$_{NoC}$ is slightly better. However, as discussed before, the performance difference induced by address translation is rather small.

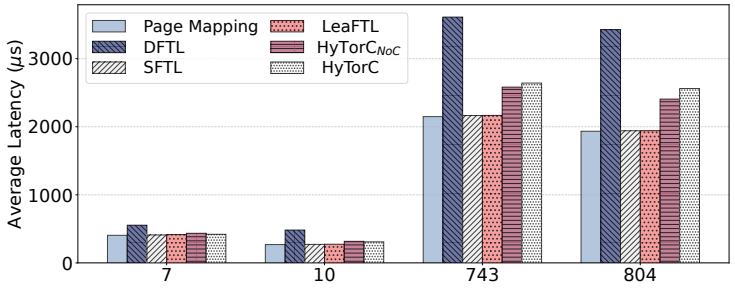


Fig. 10. Average read latencies for Alibaba cloud traces.

DFTL is significantly slower because its address translation requires additional read and write operations on a cache miss, which greatly increases write latency. DFTL is only able to provide similar performance compared to the remaining strategies for the Alibaba Cloud Trace 7, which accesses only 6% of the address space of the corresponding virtual disk, and for which DFTL can cache the entire working set.

The performance of S-FTL and LeaFTL is very similar to the performance of page mapping. Both strategies slightly increase the time to insert the new logical-physical mapping into their data structures compared to standard page mapping, which requires only a single table access. Unlike HyTorC, they do not include scrubbing, so writes do not have to queue after scrubbing operations.

4.3.2 Read Performance. Figures 10 and 11 depict the average read latencies of the four FTL schemes for the investigated traces. Due to the extremely low number of read requests in the Alibaba Cloud Trace 801, we are not showing the results for this trace.

Similar to the write request latency, the PFTL is the fastest address translation approach in all cases, as it caches all physical addresses. Also, S-FTL and LeaFTL can provide nearly the same latencies. For read-only workloads, the performance difference between page mapping and HyTorC is only based on the additional computation time of 800 ns to 1 μs of HyTorC. However, for mixed workloads, reads might again be stuck behind scrubbing operations, increasing the average read latencies. Similar to the write latencies, this can be overcome again by a more concurrent implementation that hides the scrubbing process.

4.3.3 Tail Latencies. Figure 12 presents the latency distribution for the Alibaba Trace workload. The use of HyTorC does not inherently increase tail latency. However, read and write requests may experience delays if they are queued behind the scrubbing process, leading to higher latency. These latency increases can be mitigated by employing a concurrent scrubbing mechanism, as explored in prior work for GC [11, 16, 38, 41], though implementing such a solution is outside the scope of this study.

4.4 Write Amplification

The previous evaluations have shown that the qualitative differences between the different FTL schemes remain the same regardless of whether the FTL schemes are run on the Alibaba Cloud traces or the MSR traces. Therefore, we will limit the discussion in the following to an analysis based on the Alibaba Cloud traces and will not show the corresponding results for the MSR traces.

Figure 13 shows the overall write amplification for data and metadata writes for the Alibaba traces. We have combined HyTorC and HyTorC$_{NoC}$ into a single category because their storage access patterns are nearly identical and the main difference between them is their in-memory

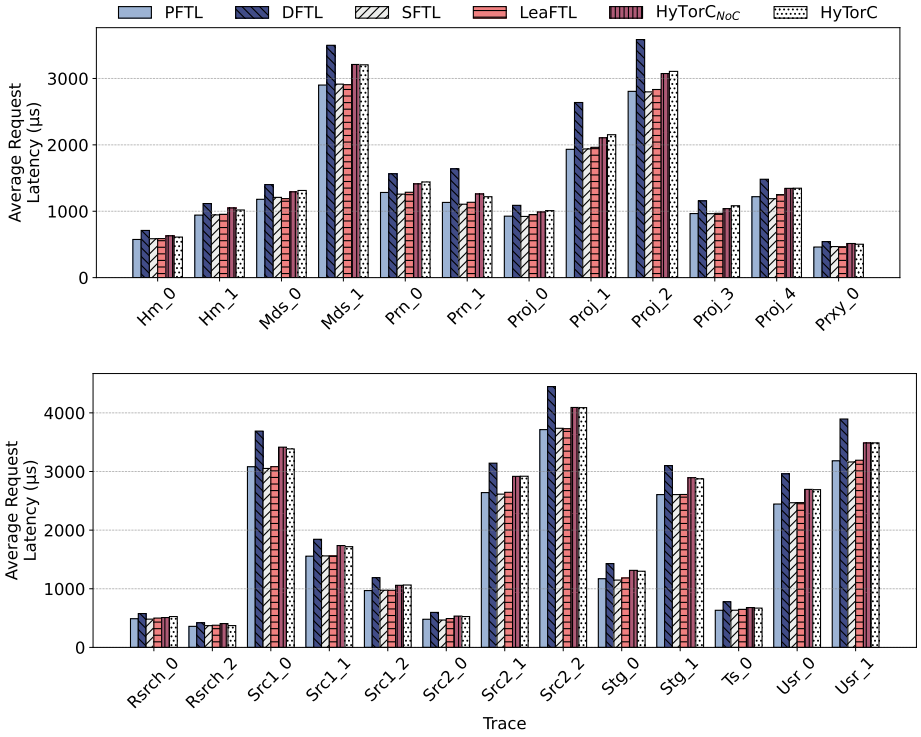


Fig. 11. Average read latencies for MSR traces.

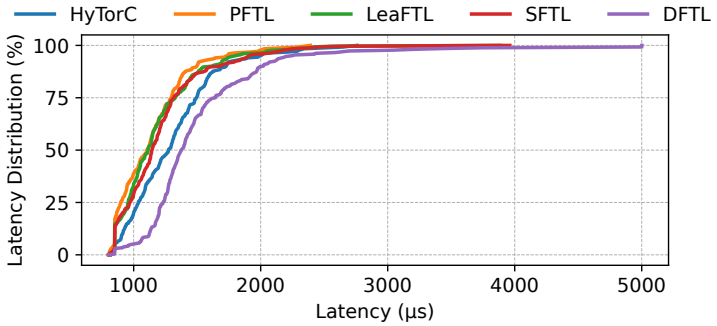


Fig. 12. Latency distribution for Alibaba cloud traces.

format. However, we distinguish between HyTorC with its default scrubbing frequency of $f_{scr} = 10$ and HyTorC$_{scr_low}$ with a lower scrubbing frequency of $f_{scr} = 5$.

PFTL, S-FTL, and LeaFTL can maintain low write amplification for each trace because they do not induce additional flash writes other than standard writes and GC writes. The write amplification of DFTL is related to the WSS. For example, for Trace 7, the working set is only 6% of the total disk capacity, so DFTL can cache almost all mapping entries that need to be used, and its write amplification is very low. Of course, the write amplification of all FTLs also depends on the write

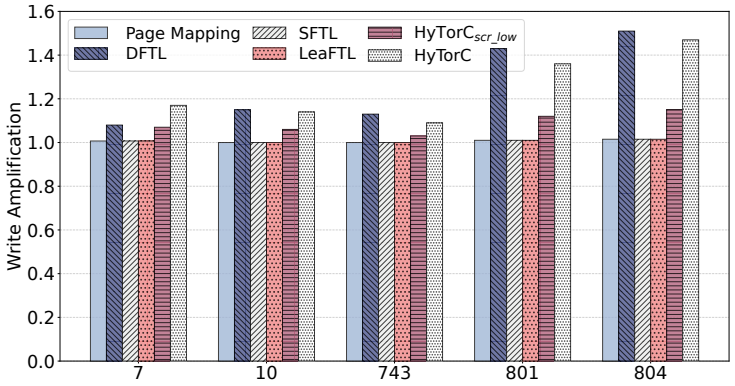


Fig. 13. Write amplification for Alibaba cloud traces.

Table 3. Data Size at Different Compression Rates (GB)

Compression Ratio	1.0x	2.75x	5.0x
After compression	2,153	853	450
Actual writes	2,512	1,929	1,788

intensity of the traces. For example, the write amplification for Trace 804, which writes the most data, is slightly higher than for the other traces.

We observe that HyTorC has a higher write amplification than PFTL, S-FTL, and LeaFTL for all traces, and sometimes a slightly higher write amplification than DFTL. This is because HyTorC's scrubbing runs continuously in the background and requires additional writes. The results for HyTorC$_{scr_low}$ show that HyTorC's write amplification can be controlled by setting its scrubbing frequency, providing a tradeoff between memory consumption and write amplification (see also Section 4.7).

However, the increase in write amplification caused by HyTorC has almost no negative impact on the lifetime of future 1 PByte SSDs. Assuming that future flash cells can withstand 250 erase cycles, which is only a quarter of today's quad-level cells, and a sustained write rate of 1 GB/s, which is extremely high for the intended workloads, the lifetime based on erase cycles would still be over five years.

4.5 Memory Usage with Data Compression

In this subsection, we present results focusing on data compression for three different synthetically generated datasets: uncompressed (compression ratio of 1.0), with an average compression ratio of 2.75, and with an average compression ratio of 5.0. We used the normal distribution to calculate the compression ratio for each block according to the target compression ratio (μ = average compression ratio; σ^2 = 1.0 or σ^2 = 3.0) and then stored only the required number of bytes based on the calculated compression ratio for a page.

Table 3 shows that the amount of data written to the SSD can be significantly higher than the compression ratio. The reason is that two or more logical pages can only be stored in a single physical page if they fit completely inside it. This limitation is implemented by many SSDs that support data compression to ensure that each page access can be served by a single I/O and is independent of the address translation mechanism (see also Section 2.3).

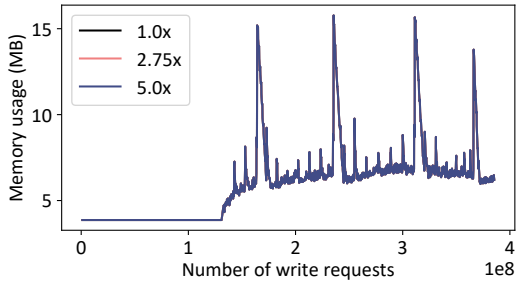


Fig. 14. Memory usage for compressed data for HyTorC.

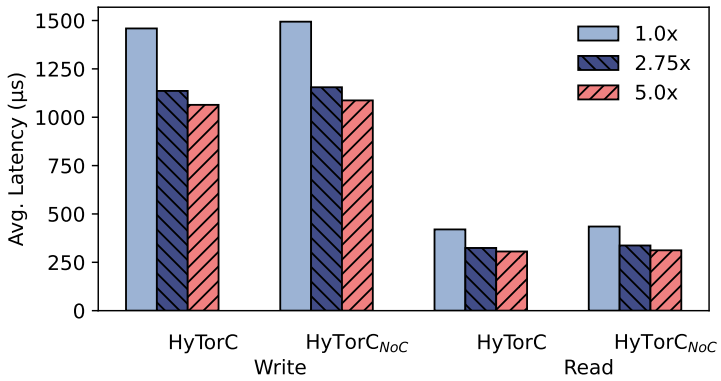


Fig. 15. Average write and read latencies for compressed data.

Data compression has no impact on the memory consumption of PFTL and DFTL. Also, it cannot be directly supported by S-FTL without introducing mechanisms similar to our compression maps. We therefore focus on HyTorC and LeaFTL in the following.

Different compression ratios have little impact on the memory footprint of HyTorC and we therefore do not repeat the numbers from Figure 5. This is not surprising for HyTorC$_{NoC}$, as the length of its compression bitmaps `cm1` and `cm2` are independent of their content. The results for HyTorC with run-length encoding show that it can also successfully compress the bitmaps, independent of the data compression ratio (see Figure 14 for Trace 7).

The memory footprint for LeaFTL depends on the compression ratio and its standard deviation. We have extended LeaFTL to also detect a slope of 1/2 logical page (or less) between physical pages, but LeaFTL still has to encode a new linear function for each change in the number of compressed logical pages per physical page, and the number of these changes increases with both the compression ratio and its standard deviation. On average, the footprint increases by 20% for a compression ratio of 2.75 and by 24% for a compression ratio of 5 if the standard deviation is 1.0. When the standard deviation is 3.0, the average memory footprint increases by 30% and 32%, respectively.

Figure 15 shows the average write and read latencies for HyTorC and HyTorC$_{NoC}$ for Trace 7. It can be seen that these latencies benefit from compression because the number of page writes decreases.

4.6 Microbenchmarks

To evaluate the robustness and adaptability of our system under varying access patterns, we designed five synthetic trace groups with 100%, 75%, 50%, 25%, and 0% random access. The

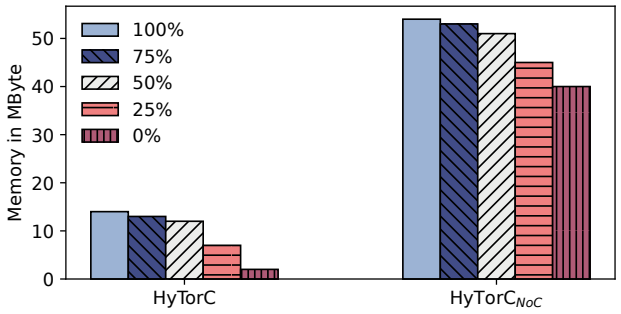


Fig. 16. Memory usage for Microbenchmarks.

non-random portion of each trace follows a sequential access pattern, allowing us to model workloads ranging from fully sequential to completely random. The capacity of the SSD in all experiments is set to 500 GByte, and each trace accesses a total of 500 GByte of data. Before running the experiments, a sequential write is performed on the entire SSD to simulate a real, fully allocated storage state. We focus on two key metrics: memory usage and average request latency, which together reflect both the efficiency and responsiveness of the system.

As shown in Figure 16, when the access pattern is completely sequential (0% random), the memory usage is the lowest. This is because HyTorC directly uses the block mapping pattern to store the sequentially accessed parts. As the proportion of random accesses increases, the memory usage also increases due to the increase in the part using page mapping. However, this growth will not continue indefinitely, and its maximum memory usage stabilizes after the scrubbing process and the increase in memory usage caused by random access have reached a balance.

The performance trends depicted in Figure 17 reflect the underlying address mapping strategy and its interaction with the background scrubbing process. For fully sequential execution, data is stored entirely in block-level mapping, avoiding the background scrubbing process and resulting in very low request latency. However, as the fraction of random accesses increases, the mapping logic gradually falls back to a fine-grained translation mechanism. This triggers the background scrubbing process, which increases latencies.

4.7 Sensitivity Analysis of the Scrubbing Parameters

The parameters of the scrubbing process described in Section 3.4 influence the memory usage of HyTorC. By setting the appropriate parameters, we can keep memory usage at a predefined lower threshold. However, configuring the parameters too aggressively can affect other performance metrics, including write amplification and latencies. In this section, we will analyze this effect by separately varying the three parameters t_{scr}, f_{scr}, and p_{scr}, while leaving the other parameters at their default values.

First, we will consider three different settings for t_{scr}, which is the minimum amount of time that must pass after writing a page belonging to an LBD before the LBD can be considered for scrubbing.

- t_{scr_none}**:** The system completely ignores the scrubbing process. Turning off scrubbing for HyTorC$_{NoC}$ is close to the S-FTL approach.
- t_{scr_low}**:** This is the default setting, in which only LBDs for which none of the logical pages has been written for more than 300 seconds are considered for scrubbing.
- t_{scr_high}**:** The parameter t_{scr} is set to 500 seconds.

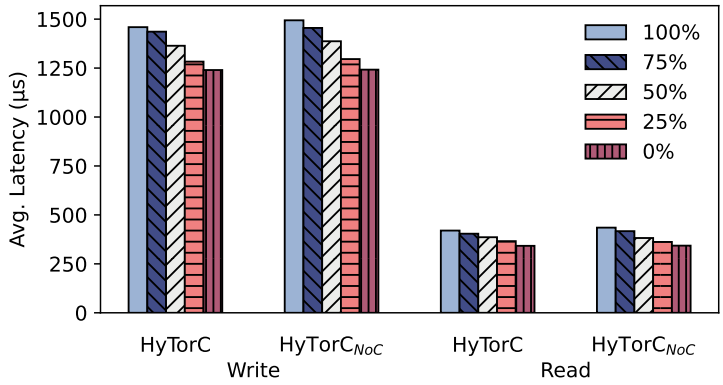
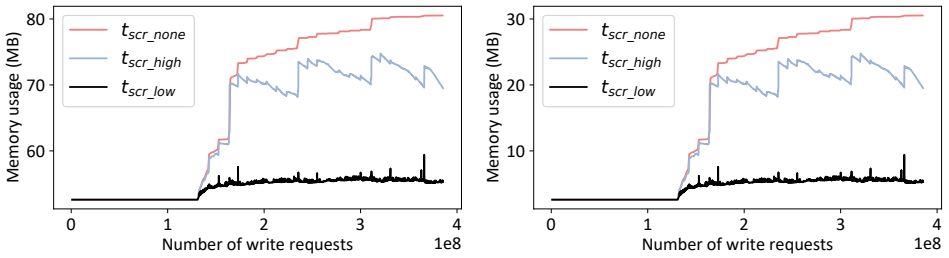


Fig. 17. Average write and read latencies for Microbenchmarks.



(a) Memory usage of HyTorC$_{NoC}$. (b) Memory usage of HyTorC.

Fig. 18. Sensitivity analysis for scrubbing parameter t_{scr} and Alibaba Trace 7.

The idea behind t_{scr} is similar to the idea behind the age parameter in the cost-benefit approach to GC [13, 30]. It ensures that logical blocks that are actively modified do not participate in the scrubbing process and do not need to be cleaned repeatedly. However, introducing t_{scr} also means that the system can never fully keep up with write updates, and setting t_{scr} to the default of 300 seconds can, therefore, result in a five minute gap between the start of writing data and the start of the scrubbing process.

This can be seen in Figure 18 for the Alibaba Trace 7. With scrubbing disabled, HyTorC and HyTorC$_{NoC}$ keep their memory footprint low during the initial sequential write process. The memory consumption then continuously grows, because both approaches are moving logical pages from being block mapped to page mapped inside their LBDs. The huge spikes in Figure 18 occur during phases of many random writes, where the block descriptors have to use a pointer per logical page, instead of being able to encode several sequentially written logical pages together by a single pointer and the corresponding bits in the sequentiality map *sm*. The shape and timing of these spikes are very similar to the spikes of the S-FTL approach seen in Figure 6 for the same workload.

Enabling scrubbing and setting $t_{scr} = 500$ results in a gap of 500 seconds between the start of Trace 7 and the scrubbing process. Any period of random writes requires some time for scrubbing to convert the LBDs back into a memory efficient representation. This time can be significantly reduced by setting $t_{scr} = 300$, its default value.

Figure 19 shows that the improved memory efficiency for a lower t_{scr} comes at the cost of a slightly increased latency. It also shows that the performance difference between HyTorC and a

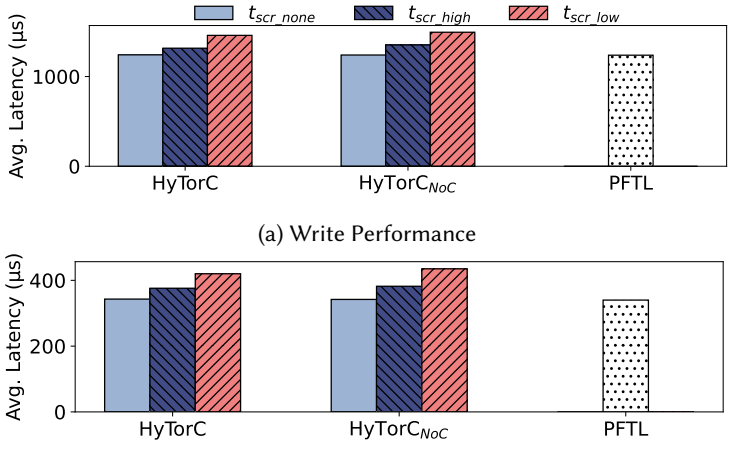


Fig. 19. Average read and write latencies for scrubbing parameter t_{scr} and Alibaba Trace 7.

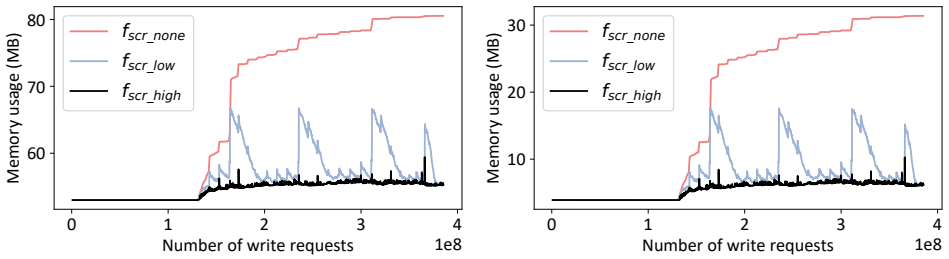


Fig. 20. Sensitivity analysis for scrubbing parameter f_{scr} and Alibaba Trace 7.

page-mapped FTL is exactly the computation time of HyTorC when scrubbing is disabled. The results show that the latency increase of HyTorC with scrubbing seen in the previous sections can be overcome by a concurrent implementation in the background.

Scrubbing wakes up f_{scr} times per second and we will now investigate the impact of the parameter f_{scr} on memory consumption and performance. We therefore again use three different settings and keep the remaining parameters at their default values:

- f_{scr_none}: The system completely ignores the scrubbing process. Turning off scrubbing for HyTorC$_{NoC}$ is close to the S-FTL approach.
- f_{scr_low}: This is a lower-frequency setting with $f_{scr} = 5$.
- f_{scr_high}: This is the baseline setting with the scrubbing frequency f_{scr} set to 10.

Figure 20(a) and (b) shows the memory consumption of HyTorC and HyTorC$_{NoC}$ for different f_{scr}. From the figures we can conclude that a higher scrubbing frequency leads to better memory efficiency. The memory consumption of f_{scr_high} is lower than that of f_{scr_low}, which is lower than that of f_{scr_none}. The figures also show that the scrubbing frequency f_{scr_high} is sufficient to keep up with the address translation changes induced by the writing process.

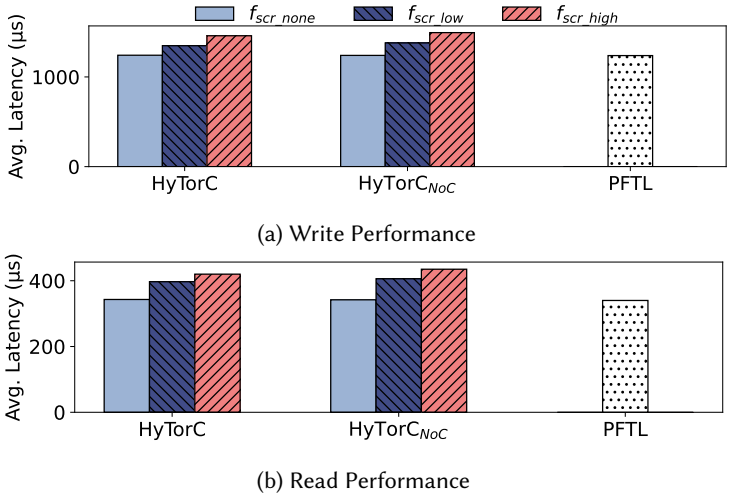


Fig. 21. Average read and write latencies for scrubbing parameter f_{scr} and Alibaba Trace 7.

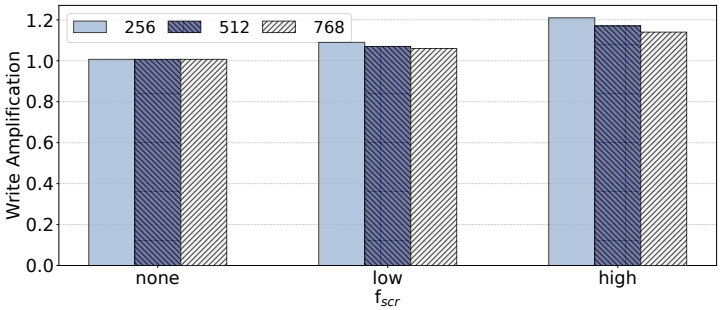


Fig. 22. Write amplification for f_{scr} and p_{scr} for Trace 7.

Figure 21 shows that the improved memory efficiency at a higher scrubbing frequency again comes at the cost of a slightly increased latency.

Finally, we investigate the impact of p_{scr} on write amplification. p_{scr} depicts the minimum number of pages of an LBD, which must be stored in page mapping mode before the block will be considered for scrubbing. For example, for Trace 7 and the default value $p_{scr} = 512$, write amplification increases from 1.007 without scrubbing (identical to PFTL) to 1.08 for low frequency and 1.17 for a high frequency scrubbing (see Figure 22). Increasing p_{scr} to 768 and only merging logical blocks that have at least 768 pages mapped in page mapping mode slightly reduces write amplification and increases write performance because fewer logical blocks are merged. This comes at the cost of an increased memory consumption. Reducing p_{scr} to 256 has the opposite effect. As discussed in Section 4.4, the slight increase in write amplification has a small impact on the lifespan of large SSDs.

The default scrubbing parameters have been chosen based on this sensitivity analysis. The optimal scrubbing parameters depend on several factors, such as read/write ratio, I/O bandwidth, and memory constraints. Developing an algorithm to automatically compute their optimal configuration will be explored in future work.

5 Conclusion

This article proposed the new SSD address translation strategy HyTorC, which aims to reduce the memory required for address translation. This method manages infrequently updated pages in block mapping mode and frequently updated pages in page mapping mode. HyTorC's metadata scheme supports flexible grouping of logical pages and the storage of multiple compressed logical pages within a single physical page. HyTorC also compresses portions of its own metadata.

HyTorC and its scrubbing process significantly reduce the memory required for address translation over the state-of-the-art, while achieving similar performance to page mapping. As future work, we will investigate an automatic scrubbing parameter selection, a fully concurrent implementation of the read, write, GC, and scrubbing processes, an even higher compression ratio by exploiting the interplay between bitmaps and block mapping, and we will investigate caching strategies for HyTorC for extremely memory-sensitive scenarios.

Acknowledgments

We would like to thank the anonymous reviewers for their helpful comments and suggestions.

References

[1] Ernst Althaus, Petra Berenbrink, André Brinkmann, and Rebecca Steiner. 2022. On the optimality of the greedy garbage collection strategy for SSDs. In *Proceedings of the 42nd IEEE International Conference on Distributed Computing Systems (ICDCS), Bologna, Italy, July 10-13*. 78–88. DOI: https://doi.org/10.1109/ICDCS54860.2022.00017

[2] Werner Bux and Ilias Iliadis. 2010. Performance of greedy garbage collection in flash-based solid-state drives. *Perform. Evaluation* 67, 11 (2010), 1172–1186. DOI: https://doi.org/10.1016/j.peva.2010.07.003

[3] Li-Pin Chang. 2007. On efficient wear leveling for large-scale flash-memory storage systems. In *Proceedings of the 2007 ACM Symposium on Applied Computing (SAC), Seoul, Korea, March 11-15*. 1126–1130. DOI: https://doi.org/10.1145/1244002.1244248

[4] Xubin Chen, Ning Zheng, Shukun Xu, Yifan Qiao, Yang Liu, Jiangpeng Li, and Tong Zhang. 2021. KallaxDB: A table-less hash-based key-value store on storage hardware with built-in transparent compression. In *Proceedings of the 17th International Workshop on Data Management on New Hardware (DaMoN), 21 June, Virtual Event, China*. 3:1–3:10. DOI: https://doi.org/10.1145/3465998.3466004

[5] Aayush Gupta, Youngjae Kim, and Bhuvan Urgaonkar. 2009. DFTL: A flash translation layer employing demand-based selective caching of page-level address mappings. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), Washington, DC, USA, March 7-11*. 229–240. DOI: https://doi.org/10.1145/1508244.1508271

[6] Erich F. Haratsch. 2019. *SSD with Compression: Implementation, Interface and Use Case*. Presentation at Flash Memory Summit, Santa Clara, CA, August 6-8.

[7] Yang Hu, Hong Jiang, Dan Feng, Lei Tian, Shu Ping Zhang, Jingning Liu, Wei Tong, Yi Qin, and Liuzheng Wang. 2010. Achieving page-mapping FTL performance at block-mapping FTL cost by hiding address translation. In *Proceedings of the IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST), Lake Tahoe, Nevada, USA, May 3-7*. 1–12. DOI: https://doi.org/10.1109/MSST.2010.5496970

[8] Shehbaz Jaffer, Kaveh Mahdaviani, and Bianca Schroeder. 2022. Improving the reliability of next generation SSDs using WOM-v codes. In *Proceedings of the 20th USENIX Conference on File and Storage Technologies (FAST), Santa Clara, CA, USA, February 22-24*. 117–132.

[9] Song Jiang, Lei Zhang, XinHao Yuan, Hao Hu, and Yu Chen. 2011. S-FTL: An efficient address translation for flash memory by exploiting spatial locality. In *Proceedings of the IEEE 27th Symposium on Mass Storage Systems and Technologies (MSST), Denver, Colorado, USA, May 23-27*. 1–12. DOI: https://doi.org/10.1109/MSST.2011.5937215

[10] Ziyang Jiao, Janki Bhimani, and Bryan S. Kim. 2022. Wear leveling in SSDs considered harmful. In *14th ACM Workshop on Hot Topics in Storage and File Systems (HotStorage), Virtual Event, June 27 - 28*. 72–78. DOI: https://doi.org/10.1145/3538643.3539750

[11] Myoungsoo Jung, Ramya Prabhakar, and Mahmut T. Kandemir. 2012. Taking garbage collection overheads off the critical path in SSDs. In *Proceedings of the ACM/IFIP/USENIX 13th International Middleware Conference (Middleware), Montreal, QC, Canada, December 3-7*. 164–186. DOI: https://doi.org/10.1007/978-3-642-35170-9_9

[12] Jeong-Uk Kang, Heeseung Jo, Jinsoo Kim, and Joonwon Lee. 2006. A superblock-based flash translation layer for NAND flash memory. In *Proceedings of the 6th ACM and IEEE International conference on Embedded software (EMSOFT), October 22-25, Seoul, Korea*. 161–170. DOI: https://doi.org/10.1145/1176887.1176911

[13] Atsuo Kawaguchi, Shingo Nishioka, and Hiroshi Motoda. 1995. A flash-memory based file system. In *Proceedings of the USENIX 1995 Technical Conference on UNIX and Advanced Computing Systems, New Orleans, Louisiana, USA, January 16-20*. 155–164.

[14] Jesung Kim, Jong Min Kim, Sam H. Noh, Sang Lyul Min, and Yookun Cho. 2002. A space-efficient flash translation layer for CompactFlash systems. *IEEE Trans. Consumer Electron.* 48, 2 (2002), 366–375. DOI: https://doi.org/10.1109/TCE.2002.1010143

[15] Hunki Kwon, Eunsam Kim, Jongmoo Choi, Donghee Lee, and Sam H. Noh. 2010. Janus-FTL: Finding the optimal point on the spectrum between page and block mapping schemes. In *Proceedings of the 10th International conference on Embedded software (EMSOFT), Scottsdale, Arizona, USA, October 24-29*. 169–178. DOI: https://doi.org/10.1145/1879021.1879044

[16] Junghee Lee, Youngjae Kim, Galen M. Shipman, Sarp Oral, and Jongman Kim. 2013. Preemptible I/O scheduling of garbage collection for solid state drives. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.* 32, 2 (2013), 247–260. DOI: https://doi.org/10.1109/TCAD.2012.2227479

[17] Sang-Won Lee, Dong-Joo Park, Tae-Sun Chung, Dong-Ho Lee, Sangwon Park, and Ha-Joo Song. 2007. A log buffer-based flash translation layer using fully-associative sector translation. *ACM Trans. Embed. Comput. Syst.* 6, 3 (2007), 18. DOI: https://doi.org/10.1145/1275986.1275990

[18] Sungjin Lee, Jihoon Park, Kermin Fleming, Arvind, and Jihong Kim. 2011. Improving performance and lifetime of solid-state drives using hardware-accelerated compression. *IEEE Trans. Consumer Electron.* 57, 4 (2011), 1732–1739. DOI: https://doi.org/10.1109/TCE.2011.6131148

[19] Sungjin Lee, Dongkun Shin, Young-Jin Kim, and Jihong Kim. 2008. LAST: Locality-aware sector translation for NAND flash memory-based storage systems. *ACM SIGOPS Oper. Syst. Rev.* 42, 6 (2008), 36–42. DOI: https://doi.org/10.1145/1453775.1453783

[20] Jinhong Li, Qiuping Wang, Patrick P. C. Lee, and Chao Shi. 2020. An In-depth analysis of cloud block storage workloads in large-scale production. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC), Beijing, China, October 27-30*. 37–47. DOI: https://doi.org/10.1109/IISWC50251.2020.00013

[21] Stathis Maneas, Kaveh Mahdaviani, Tim Emami, and Bianca Schroeder. 2022. Operational characteristics of SSDs in enterprise storage systems: A large-scale field study. In *Proceedings of the 20th USENIX Conference on File and Storage Technologies (FAST), Santa Clara, CA, USA, February 22-24*. 165–180.

[22] Fabio Margaglia and André Brinkmann. 2015. Improving MLC flash performance and endurance with extended P/E cycles. In *Proceedings of the IEEE 31st Symposium on Mass Storage Systems and Technologies, (MSST), Santa Clara, CA, USA, May 30 - June 5*. 1–12. DOI: https://doi.org/10.1109/MSST.2015.7208278

[23] Fabio Margaglia, Gala Yadgar, Eitan Yaakobi, Yue Li, Assaf Schuster, and André Brinkmann. 2016. The devil Is in the details: Implementing flash page reuse with WOM codes. In *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST), Santa Clara, CA, USA, February 22-25*. 95–109.

[24] Dushyanth Narayanan, Austin Donnelly, and Antony I. T. Rowstron. 2008. Write Off-loading: Practical power management for enterprise storage. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies (FAST), February 26-29, San Jose, CA, USA*. 253–267.

[25] Our World in Data. 2025. Historical price of computer memory and storage. (2025). Retrieved July 18, 2025 from https://ourworldindata.org/grapher/historical-cost-of-computer-memory-and-storage

[26] Youngjo Park and Jin-Soo Kim. 2011. zFTL: Power-efficient data compression support for NAND flash-based consumer electronics devices. *IEEE Trans. Consumer Electron.* 57, 3 (2011), 1148–1156. DOI: https://doi.org/10.1109/TCE.2011.6018868

[27] Pascari. 2024. *Enterprise D-Series: High Capacity PCIe Gen5 Data Center Storage Solution PASCARI D200V*. Technical Report. Pascari.

[28] Yifan Qiao, Xubin Chen, Ning Zheng, Jiangpeng Li, Yang Liu, and Tong Zhang. 2022. Closing the B+-tree vs. LSM-tree write amplification gap on modern storage hardware with built-in transparent compression. In *Proceedings of the 20th USENIX Conference on File and Storage Technologies (FAST), Santa Clara, CA, USA, February 22-24*. 69–82.

[29] Zhiwei Qin, Yi Wang, Duo Liu, and Zili Shao. 2011. A two-level caching mechanism for demand-based page-level address mapping in NAND flash memory storage systems. In *Proceedings of the 17th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), Chicago, Illinois, USA, 11-14 April*. 157–166. DOI: https://doi.org/10.1109/RTAS.2011.23

[30] Mendel Rosenblum and John K. Ousterhout. 1992. The design and implementation of a log-structured file system. *ACM Trans. Comput. Syst.* 10, 1 (1992), 26–52. DOI: https://doi.org/10.1145/146941.146943

[31] Reza Salkhordeh, Kevin Kremer, Lars Nagel, Dennis Maisenbacher, Hans Holmberg, Matias Bjørling, and André Brinkmann. 2021. Constant time garbage collection in SSDs. In *Proceedings of the IEEE International Conference on Networking, Architecture and Storage (NAS), Riverside, CA, USA, October 24-26*. 1–9. DOI: https://doi.org/10.1109/NAS51552.2021.9605386

[32] Solidigm™. 2025. *Solidigm D5-P5336 Product Brief: The World's Highest Capacity PCIe SSD*. Technical Report. Solidigm.

[33] Jinghan Sun, Shaobo Li, Yunxin Sun, Chao Sun, Dejan Vucinic, and Jian Huang. 2023. LeaFTL: A learning-based flash translation layer for solid-state drives. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), Volume 2, Vancouver, BC, Canada, March 25-29*. 442–456. DOI: https://doi.org/10.1145/3575693.3575744

[34] Reza Gholami Taghizadeh, Mohammadreza Binesh Marvasti, Seyyed Amir Asghari, Ramin Gholami Taghizadeh, Morteza Nabavi, and Yvon Savaria. 2022. IBU: An In-block update address mapping scheme for solid-state drives. *IEEE Access* 10 (2022), 4934–4947. DOI: https://doi.org/10.1109/ACCESS.2021.3138832

[35] Qiuping Wang, Jinhong Li, Patrick P. C. Lee, Tao Ouyang, Chao Shi, and Lilong Huang. 2022. Separating data via block invalidation time inference for write amplification reduction in log-structured storage. In *Proceedings of the 20th USENIX Conference on File and Storage Technologies (FAST), Santa Clara, CA, USA, February 22-24*. 429–444.

[36] Shengzhe Wang, Zihang Lin, Suzhen Wu, Hong Jiang, Jie Zhang, and Bo Mao. 2024. LearnedFTL: A learning-based page-level FTL for reducing double reads in flash-based SSDs. In *Proceedings of the IEEE International Symposium on High-Performance Computer Architecture (HPCA), Edinburgh, United Kingdom, March 2-6*. 616–629. DOI: https://doi.org/10.1109/HPCA57654.2024.00054

[37] Western Digital. 2024. *Ultrastar® DC SN655: Unlocking the Full Potential of Enterprise SSDs*. Technical Report. Western Digital.

[38] Guanying Wu and Xubin He. 2012. Reducing SSD read latency via NAND flash program and erase suspension. In *Proceedings of the 10th USENIX conference on File and Storage Technologies (FAST), San Jose, CA, USA, February 14-17*. 10.

[39] Michael Wu and Willy Zwaenepoel. 1994. eNVy: A non-volatile, main memory storage system. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), San Jose, California, USA, October 4-7*. 86–97. DOI: https://doi.org/10.1145/195473.195506

[40] Gala Yadgar, Eitan Yaakobi, Fabio Margaglia, Yue Li, Alexander Yucovich, Nachum Bundak, Lior Gilon, Nir Yakovi, Assaf Schuster, and André Brinkmann. 2018. An analysis of flash page reuse with WOM codes. *ACM* Trans. Storage 14, 1 (2018), 10:1–10:39. DOI: https://doi.org/10.1145/3177886

[41] Shiqin Yan, Huaicheng Li, Mingzhe Hao, Michael Hao Tong, Swaminathan Sundararaman, Andrew A. Chien, and Haryadi S. Gunawi. 2017. Tiny-tail flash: Near-perfect elimination of garbage collection tail latencies in NAND SSDs. *ACM Trans. Storage* 13, 3 (2017), 22:1–22:26. DOI: https://doi.org/10.1145/3121133

[42] Chi Zhang, Yi Wang, Tianzheng Wang, Renhai Chen, Duo Liu, and Zili Shao. 2014. Deterministic crash recovery for NAND flash based storage systems. In *Proceedings of the 51st Annual Design Automation Conference (DAC), San Francisco, CA, USA, June 1-5*. 148:1–148:6. DOI: https://doi.org/10.1145/2593069.2593124

[43] Ning Zheng, Xubin Chen, Jiangpeng Li, Qi Wu, Yang Liu, Yong Peng, Fei Sun, Hao Zhong, and Tong Zhang. 2020. Re-think data management software design upon the arrival of storage hardware with built-in transparent compression. In *Proceedings of the 12th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage), July 13-14*.

[44] You Zhou, Fei Wu, Ping Huang, Xubin He, Changsheng Xie, and Jian Zhou. 2015. An efficient page-level FTL to optimize address translation in flash memory. In *Proceedings of the 10th European Conference on Computer Systems (EuroSys), Bordeaux, France, April 21-24*. 12:1–12:16. DOI: https://doi.org/10.1145/2741948.2741949

[45] Aviad Zuck, Sivan Toledo, Dmitry Sotnikov, and Danny Harnik. 2014. Compression and SSDs: Where and How?. In *Proceedings of the 2nd Workshop on Interactions of NVM/Flash with Operating Systems and Workloads (INFLOW), Broomfield, CO, USA, October 5*.

Received 27 January 2025; revised 23 July 2025; accepted 28 July 2025