Fully Harnessing the Performance Potential of DRAM-less Mobile Flash Storage

Jaesun No[1,2]
Samsung Electronics
Suwon, Republic of Korea
jaesun.no@csi.skku.edu

Gyusun Lee[2]
Sungkyunkwan University
Suwon, Republic of Korea
gyusun.lee@csi.skku.edu

Youngsok Kim
Yonsei University
Seoul, Republic of Korea
youngsok@yonsei.ac.kr

Jinkyu Jeong[3]
Yonsei University
Seoul, Republic of Korea
jinkyu@yonsei.ac.kr

***Abstract*—Mobile flash-based storage devices, such as universal flash storage (UFS), provide high-performance block I/O services in mobile systems. However, the performance of mobile flash storage is yet limited as such devices cannot afford a large internal DRAM due to various constraints, such as limited form factor, power/cost budget, etc. Host Performance Booster (HPB) has been proposed to overcome this performance limitation by borrowing and utilizing the rich host DRAM. Nonetheless, HPB is yet suboptimal since it accelerates only read I/Os and is oblivious to write I/Os and flash-memory-inherent operations, such as garbage collection (GC) and wear-leveling.**

In this paper, we propose high-performance UFS (HP-UFS) that fully harnesses the performance potential of DRAM-less mobile flash storage devices. Unlike previous work, HP-UFS accelerates all types of storage operations from read/write I/Os to GC and wear-leveling. HP-UFS borrows but proactively manages the host DRAM to accelerate all the storage operations. HP-UFS preserves storage metadata, such as logical-to-physical mapping, and valid page bitmap/count up-to-date in the host DRAM; these metadata help to improve the storage performance by reducing the overheads of accessing these metadata. Through our experimental evaluations with various micro-benchmark workloads, we demonstrate the superior performance of HP-UFS in terms of random I/O and GC. With real-world workload scenarios, the proposed HP-UFS device outperforms HPB by up to 39% (without GC) and 88% (with GC), and shows only a 4% average performance gap compared to an ideal flash storage device with plenty of DRAM.

***Index Terms*—mobile flash storage, universal flash storage, host performance booster, garbage collection**

I. Introduction

Flash memory-based mobile storage devices, such as embedded MultiMediaCard (eMMC) [1] and universal flash storage (UFS) [2], have driven the advancement of mobile and embedded computing systems. As compared to traditional hard disk drives, flash memory offers various attractive benefits for mobile computing systems, such as high performance, high shock resistance, low power consumption, small form factor, etc. These benefits allow flash memory to become the secondary storage of mobile computing systems, such as smartphones, tablet personal computers. Mobile and tablet devices have already surpassed desktop computers in terms of market share [3]. Alongside smartphones, other devices like smartwatches, auto parts, and home automation devices also utilize mobile flash storage devices.

Mobile flash storage differs from that of desktop or server systems in that mobile flash storage is DRAM-less [1], [2], [4], [5]. Mobile systems are limited in terms of form factor, power consumption, and manufacturing cost. Therefore, mobile flash storage devices are not easy to afford a large capacity DRAM installed. Unfortunately, without DRAM, mobile flash storage cannot deliver its full performance potential [4]–[7]. The flash translation layer (FTL) needs to manage necessary metadata in DRAM for high performance. Such metadata include a logical-to-physical (L2P) mapping table and valid page bitmap and counter [8]. These metadata are accessed frequently while performing storage operations, such as read/write I/Os, garbage collection (GC), and wear-leveling. When these metadata are stored in fast memory, such as DRAM, the storage performance can be enhanced. However, when stored in slow memory, such as flash memory, their access overhead can negatively impact and thus degrade storage performance [4], [5]. SRAM is available in most flash-based storage devices but its benefit for metadata access is limited due to its small capacity (e.g., a few MBs). Once the working set of metadata exceeds the capacity of the SRAM, their access overhead dominates the storage performance [4], [5], [9].

To address the absence of DRAM in mobile flash storage, one compelling approach is borrowing and utilizing host DRAM (e.g., unified memory extension (UME) [10], [11] and host performance booster (HPB) [5], [12]). In these approaches, the host DRAM is used to expand the cache of L2P mappings (in-device SRAM + a portion of host DRAM). Consequently, storage operations can be accelerated by storing and accessing the metadata in fast memory (host DRAM or in-device SRAM). Among the two methods, HPB is considered practical since it does not require hardware modification, and all of its features can be implemented in software (the host device driver and the FTL firmware). In practice, HPB is adopted by a commercial product, the Google Pixel 3 smartphone in 2018 with the Linux kernel version 4.9.96 [4], [13]. In HPB, the host device driver and the FTL firmware exchange necessary data by piggybacking those in UFS command/response pairs [12], [13].

While HPB has demonstrated good performance with read-intensive workloads [4], [5], it is incomplete in addressing

[1]This work was done when he was at Sungkyunkwan University.
[2]These authors contributed equally to this work.
[3]Jinkyu Jeong is the corresponding author.

all of the benefits of utilizing host DRAM. Especially, HPB is oblivious to write I/Os, GC and wear-leveling operations. In mobile workloads, write I/Os are also important as they affect the user latency of data persistence operations (e.g., `fsync()`) [14]–[17]. Also, slow handling of write I/Os may affect read I/Os due to the contention in the device. In addition, flash memory GC can cause unexpected long delays because it can increase the latency of handling forthcoming block I/O requests [18]–[20]. Without the support for these operations, the use of host DRAM is suboptimal and is not able to deliver the full performance potential of the underlying flash memory. For example in UFS or HPB, when GC occurs, the flash memory is busy reading L2P mappings for checking the validness of pages. This not only delays the completion of GC but also incurs contention with foreground activities. These overheads ultimately lead to a degraded user experience due to worsen responsiveness of mobile systems.

In this paper, we propose high-performance UFS (HP-UFS) that fully harnesses the performance potential of DRAM-less flash-based mobile storage devices. HP-UFS utilizes the host DRAM to accelerate not only read I/Os but also write I/Os and even GC and wear-leveling operations. To this end, the host device driver and the FTL firmware work in harmony to exchange necessary metadata. Especially, HP-UFS aggressively caches modified L2P mappings in the host DRAM. This allows the opportunity to save the flash memory bandwidth which should have been used for L2P mapping persistence operations. In addition, the aggressive caching of modified L2P mappings allows to manage the two important metadata structures, valid page bitmap and valid page count in the host DRAM. This allows HP-UFS to reduce the overheads of the flash memory GC and wear-leveling operations, once otherwise the flash memory bandwidth should be consumed for identifying the validness of pages involved in GC. At the same time, HP-UFS preserves the crash consistency of modified L2P mappings by its carefully-designed L2P mapping writeback mechanism. Finally, HP-UFS adopts a simple yet effective least-recently-used (LRU) replacement for cached L2P mappings to preserve the metadata working-set in the host DRAM while keeping the host DRAM usage low.

We implemented the HP-UFS prototype using the FEMU SSD emulator [21]. We configured the flash memory architecture and low-level parameters to build a UFS emulator, which shows almost identical performance to real UFS 3.0 hardware [22]. Then, we implemented the prototype of HP-UFS upon the UFS emulator and the HP-UFS host device driver in the Linux kernel. The evaluation results using the Flexible I/O tester (FIO) [23] micro-benchmark show that HP-UFS outperforms HPB by up to 78% and 70% in terms of random write performance without GC and with GC, respectively. To measure the impact of HP-UFS on user experience, we have tested the performance of SQLite database. HP-UFS outperformed HPB in terms of SQL transaction latency [24]; HP-UFS has shown up to 88% latency reduction when GC occurs, and without GC, HP-UFS still has shown up to 39% latency reduction. The trace-driven evaluation of mobile application workloads also has shown performance improvement by HP-UFS. Especially, with real-world workloads, HP-UFS has shown only 4% performance degradation as compared to an ideal UFS device with plenty of DRAM. These results indicate that HP-UFS almost fully harnesses the performance potential of the DRAM-less mobile flash storage.

This paper has the following contributions:

- We identify the performance problems of existing mobile DRAM-less storage that utilizes the host DRAM (i.e., UME [11] and HPB [5]).
- We propose a practical approach to accelerate all types of storage operations (read, write, GC, and wear-leveling) in the DRAM-less mobile flash storage.
- We demonstrate the effectiveness of the proposed scheme by evaluating our scheme on both synthetic and real-world mobile workloads.

II. Background and Motivation

A. Background

Metadata in Flash-based Storage Systems. Flash memory provides various advantages, such as high performance, high shock resistance, etc., for realizing and improving mobile computing systems. However, flash memory has inherent limitations (e.g., erase-before-write and asymmetric I/O unit), which prevents flash memory from replacing traditional disks. Therefore, a software layer called flash memory translation layer (FTL) plays a crucial role in aligning the storage interface of flash memory to that of disks. FTL manages an important metadata structure called *logical-to-physical (L2P) mapping table* that translates a logical page address (LPA) seen by host (e.g., file systems) to a physical page address (PPA). Although, the performance and cost of L2P mapping table change greatly depending on the mapping scheme used [8], [25]–[29], page-level mapping [26] is a de-facto standard mapping scheme because of its high performance on both sequential and random workloads as well as high GC efficiency [9], [30], [31]. In page-level mapping FTL, every 4 KB page is translated by the L2P mapping table. This makes one important disadvantage that the cost of the mapping table is high. For example, a 1 TB SSD requires a 1 GB mapping table (4 byte entry for each 4 KB page).

In addition, FTL needs to manage other metadata structures, *valid page bitmap* and *valid page count* for efficient GC and wear-leveling operations. The *valid page bitmap* keeps track of the validity of all physical pages in flash memory. Hence, the bitmap is used to identify pages to preserve and pages to erase during GC. Similarly, the *valid page count* keeps track of the number of valid pages in each block, aiding in the selection of a victim block for GC. These metadata structures play a crucial role in enhancing the efficiency of GC and wear-leveling operations thereby improving the overall performance of flash-based storage systems [32].

Metadata Management in DRAM-less Mobile Storage. In flash-based storage systems, managing such metadata structures in fast memory, such as DRAM, is important in storage performance. Desktop or enterprise SSDs can afford a large

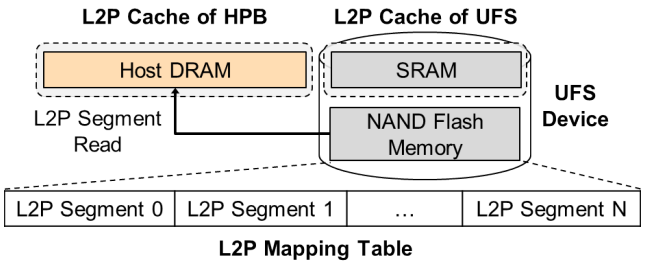


Fig. 1. Overview of L2P mapping management in DRAM-less mobile storage (UFS and HPB).

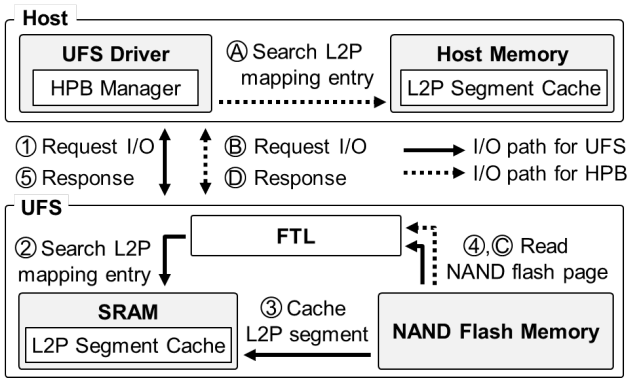


Fig. 2. Overview of read I/O path for DRAM-less mobile storage.

capacity DRAM considering its form factor and power/cost budgets, so they can store and access these metadata in in-device DRAM. However, mobile flash storage devices, such as UFS and eMMC, are DRAM-less and have only a limited-capacity SRAM (of a few hundred kilobytes [33]). Various constraints, such as form factor, power/cost budget, etc., prevent the installation of a large-capacity DRAM in mobile flash storage.

Given the limited-size SRAM and the absence of DRAM in UFS, storing the entire L2P mapping table in memory is not feasible. Therefore, FTLs of UFS adopt a caching approach, where only a portion of the mapping table is cached in SRAM [9], [12]. Figure 1 illustrates this concept. The L2P mapping table is divided into multiple segments, with each segment occupying a flash memory page. Typically, each L2P segment is 4 KB in size, accommodating 1,024 L2P mapping entries of 4 bytes each. When FTL processes an I/O request from the host, it checks whether necessary L2P mapping entries are cached in SRAM. If not, the FTL needs to fetch the missing L2P entries from the flash memory and then handle the host requests. Therefore, the storage performance is largely affected by the caching behavior; sequential access shows decent performance while random access shows poor due to high L2P mapping cache misses followed by slow flash memory accesses [7], [22].

Borrowing Host DRAM for Mobile Flash Storage. The state-of-the-art approach to overcoming the absence of DRAM in mobile flash storage is utilizing the large-capacity host DRAM. In Figure 1, a portion of host DRAM is used to cache L2P mapping entries thereby improving the L2P cache hit ratio. As a result, storage performance can be improved by avoiding costly flash memory access for retrieving L2P mappings. For UFS mobile storage, unified memory extension (UME) [10], [11], and host performance booster (HPB) [5], [12] have been proposed to utilize host DRAM for expanding the L2P mapping cache of FTL. Among them, HPB is considered practical since it requires no modification of existing hardware, and all the features can be implemented in software, the host device driver and the FTL firmware. Therefore, in 2018, Google's Pixel 3 was equipped with HPB-enabled UFS with the HPB device driver in Linux kernel 4.9.96 [13].

Figure 2 illustrates how UFS and HPB handle host I/O requests. Initially, in UFS, when the host-side UFS driver receives an I/O request, it dispatches the request to the UFS device (① in the figure). The FTL within the UFS device then performs the necessary address translation of LPA to PPA. To accomplish this, the FTL searches for the required L2P mapping entry from the L2P segment cache stored in SRAM (②). If the L2P entry is not present in the cache, the FTL retrieves the corresponding L2P segment from the NAND flash memory and caches it in the SRAM (③). Subsequently, it proceeds to read the relevant NAND flash page using the retrieved PPA (④) and transmits the page to the host (⑤).

In HPB, when the device driver receives an I/O request, it checks the HPB host memory cache that stores L2P segments first (Ⓐ). If the required L2P mapping is found, the I/O request is tagged with the corresponding PPA and sent to the HPB device (Ⓑ). The FTL then uses the received PPA to access the NAND flash memory directly without any address translation step (Ⓒ). Finally, it transfers the response back to the host (Ⓓ). Consequently, HPB can avoid one (slow) flash memory access during the read operation as compared to UFS, when the required L2P mapping is found from L2P mapping cache in the host memory. On the other hand, when a desired L2P mapping is not found, the HPB driver initiates an *L2P segment read command* to read and cache an L2P segment from the (slow) flash memory. When this L2P cache miss occurs, the read latency would be worse than the original UFS. However, HPB may show better read performance than UFS because of using a larger memory for the L2P mapping cache.

B. Motivation

HPB has improved the read performance significantly, by utilizing the host DRAM, which in turn reduces L2P cache misses [4], [5]. However, it lacks accelerating other major storage operations, such as writes, GC and wear-leveling. In mobile workloads, not only reads but also writes are important for user experience [14]–[17], [34]–[38]. Moreover, when GC occurs in flash-based storage, it significantly aggravates the user experience due to GC-induced long latencies in storage operations [18], [19], [39]. The idea of borrowing and utilizing the rich host DRAM is vital for DRAM-less mobile flash storage systems. At the same time, it is necessary to consider and accelerate all types of storage operations.

Specifically, HPB unnecessarily pollutes flash memory bandwidth because of its partial support of storage operations. Whenever L2P mappings are modified, HPB invalidates the

cached L2P segment in the host memory [5], [12]. Subsequent reads to the same LBA or adjacent LBAs within the invalidated L2P segment experience a long delay of fetching the L2P segment from flash memory [4]. Since L2P segment invalidation can occur in many ways, such as host writes, GC and wear-leveling, it may unexpectedly affect user experience by increasing tail latency. This extra L2P segment retrieval can be caused by not only host write request but also any operations modifying L2P mappings (e.g., GC and wear-leveling).

The change of L2P mappings incurs more than the L2P invalidation-related costs in DRAM-less flash storage systems. Whenever L2P mappings are modified, changes should be persisted in the flash memory. Since mobile flash storage has a limited amount of buffers for L2P mappings, L2P segments are frequently read, modified and written back to reflect changed L2P entries. Considering the limitation of form factor, power/cost budgets of mobile storage, solutions like provisioning super capacitors [40] or other non-volatile memory are impractical [41]. HPB has exploited the rich host DRAM but still limits the use of the host DRAM as a cache of L2P mappings and does not accommodate modified L2P mappings (i.e., no L2P mapping buffering). Therefore, with write-intensive workloads, the flash memory bandwidth can be largely consumed by the read-modify-write of L2P segments.

Finally, DRAM-less flash storage systems experience the hardship of performing GC and wear-leveling operations. These operations are designated to keep the flash memory healthier and to adhere to the erase-before-write characteristic. Their essential internal operation is identifying the validity of pages. Without DRAM, it is impractical to manage valid page bitmap per each flash block because this metadata size overfits the available SRAM of mobile flash storage. For example, a 1 TB SSD requires 128 MB of valid page bitmap, assuming one bit per 4 KB page. It cannot fit in a few hundred KBs SRAM of mobile flash storage [33]. As an alternative, the GC operation identifies the validity of pages by inspecting the spare area of flash page and the L2P mapping table; if a flash page is valid, its PPA should be the value of the corresponding LPA in the L2P mapping table. The problem in DRAM-less flash storage system is that the L2P mapping is not in fast memory but in slow flash memory. Therefore, GC and wear-leveling operations incur significant overhead of reading L2P segments from flash memory.

Consequently, borrowing and utilizing the rich host DRAM is a reasonable approach to overcoming the limitation of DRAM-less mobile storage devices. However, current solutions support only read operations and lack supporting write and GC/wear-leveling operations, therefore cannot fully harness the performance potential of all the flash memory hardware installed.

III. Design of HP-UFS

A. Overview

We propose HP-UFS (high-performance UFS) that fully harnesses the performance potential of mobile flash-based storage devices. Similar to previous work, HP-UFS borrows and utilizes the rich host DRAM. However, different from previous approaches that have focused only on improving read performance, HP-UFS accelerates all I/O operations in mobile flash-based storage devices, hence achieving high performance.

HP-UFS utilizes the host DRAM in a more proactive way than the previous work [5]. In the previous work, whenever an L2P mapping is modified due to flash memory writes by host write request, GC or wear-leveling, the cached L2P segment is invalidated from host memory. Then, any forthcoming I/O requests require additional (slow) flash memory access to retrieve the invalidated L2P segment again, which not only prolongs the latency of I/O requests but also incurs bandwidth contention on flash memory chip/channels. However in HP-UFS, whenever L2P mappings are changed by flash memory writes, the changed L2P mapping information is actively propagated to the host DRAM. Consequently, the host can have its cached L2P mappings up-to-date and therefore, can avoid costly L2P mapping retrieval from flash memory [5].

HP-UFS's proactive L2P mapping management not only reduces overheads associated with invalidating and fetching up-to-date L2P mappings but also allows new opportunities to manage valid page bitmap and valid page count in the host DRAM. Since the valid page bitmap is too large to fit in the SRAM of mobile flash storage, the FTL had no choice but to perform GC and wear-leveling inefficiently as explained in Section II-B. In HP-UFS, however, whenever L2P mappings are changed, the changed information is proactively propagated from the flash storage device to the host DRAM. Consequently, the host DRAM can also manage valid page bitmap and valid page count by utilizing the L2P change information received from the storage device. By using these two metadata structures, GC and wear-leveling operations can be performed with low cost; flash memory accesses are unnecessary during the page validity identification. As a result, HP-UFS can perform GC as fast as the flash-based storage systems with plenty of DRAM (e.g., desktop or enterprise SSDs). Therefore, the tail latency of user experience can be greatly reduced because of reduced GC/wear-leveling operation time.

To achieve the benefits mentioned above, HP-UFS faces two important challenges: (1) preserving the consistency of L2P mappings between the host DRAM and the storage device while keeping the performance high, and (2) allowing necessary data exchange between the host device driver and the storage device under the limitations of the UFS protocol [2]. The first challenge is crucial because if any mapping changes made in the storage device are synchronously propagated to the host driver, the consistency is kept high but the storage performance will be degraded due to waiting times enforced by synchronous data exchange. Hence, it is necessary to allow the host driver and the storage device to perform independently while preserving the consistency of L2P mappings. HP-UFS carefully address this by introducing a *sequence number*. All new L2P mappings are assigned a sequence number and relevant data structures also inherit this. The sequence number is used to identify up-to-date L2P mappings or other metadata.

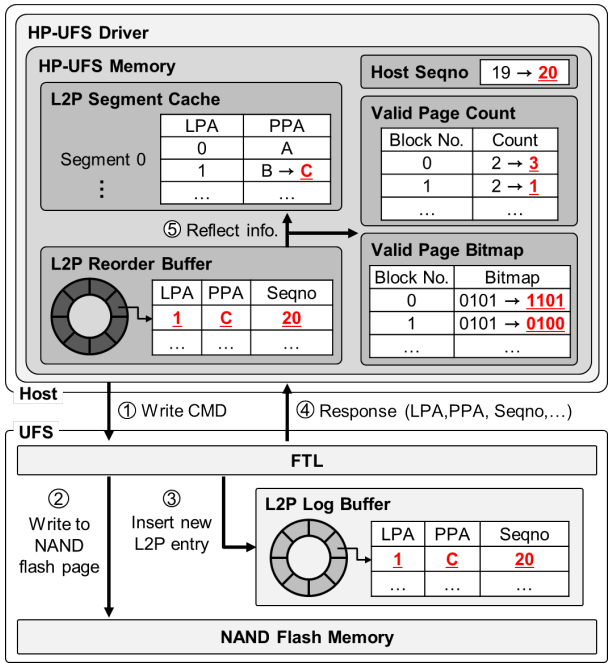


Fig. 3. The overall design and sequences of host write request in HP-UFS.

This allows buffering and out-of-order transmission of L2P modification information.

The second challenge is also important in devising a scheme for mobile storage systems. The UFS protocol is a de-facto standard mobile storage interface providing high-performance non-volatile storage with low power consumption and form factor [10]. It is not advisable to implement a radical shift in storage hardware or protocols. Hence, the proposed scheme can be implemented upon the exploitation of the existing protocol (i.e., UFS) while expanding the ability to utilize the host DRAM for accelerating storage operations. HP-UFS exploits the piggybacking of information in the command/response of the UFS protocol, which is already exploited in previous work [4], [5]. HP-UFS introduces only a few more commands to facilitate the metadata exchange between the device driver in the host and the device FTL firmware.

B. Handling Host Write Request

When a write I/O request arrives at the HP-UFS driver, it first checks whether the associated L2P segment is cached in the L2P segment cache. If not, the driver issues an L2P segment read command first, and then sends the actual write command to the device (① in Figure 3). Here, the driver does not wait for the read of the L2P segment since write commands do not need PPAs. When the HP-UFS device controller receives a write command, it allocates a new PPA for the write and assigns a new sequence number. Then, it issues a NAND flash page write (②). After the write completes, it inserts the new L2P entry (i.e., a modified L2P mapping) into an *L2P log buffer*, which buffers modified L2P segments in in-device SRAM until those are sent to the host DRAM (③). In the example in the figure, LPA 1 is assigned new PPA C and its sequence number is 20. After that, the controller relays the L2P entry to the host driver by piggybacking it in an I/O response (④). When the host receives the L2P entry, the entry is staged temporarily in an *L2P reorder buffer*. The L2P reorder buffer is a staging area of L2P entries transferred from the HP-UFS storage device. L2P entries in the reorder buffer are not processed immediately because L2P entries can be reordered by FTL due to load imbalance between flash chips/channels, or because an L2P segment to which a modified L2P mapping entry should be applied is not cached in the L2P segment cache yet. The L2P change entries are processed in the order of sequence number and are reflected to cached L2P segments (⑤). Hence, each L2P entry is applied to the associated L2P segment, valid page bitmap and valid page count. In the figure, the L2P segment 0 modifies its mapping of LPA 1 from B to C. The valid page bitmap and count are also modified accordingly (C is the fourth page of block 0, B is the first page of block 1). At last, the *host sequence number* is updated to that of the latest L2P entry processed in the host driver. In the figure, the host sequence number is changed from 19 to 20 since the L2P mapping related to the new sequence number is applied to its L2P mapping cache.

Each host write request might be expected to be accompanied by its response with a piggybacked L2P entry, hence having 1-to-1 relationship which eases the synchronous update of L2P segment in the host DRAM for every host writes. However, the synchronous L2P segment update is impossible because a lot more L2P changes are made for several reasons and their transfer to the host DRAM can be backlogged if the response-piggybacking method is solely used.

The reason for having a lot more L2P changes is twofold: FTL-induced bulk flash writes and host write requests that span on multiple flash blocks. The former is intuitive in that these operations produce many L2P entries. The latter is made by one host write request with a large I/O size and if a current block cannot host the entire write request, another flash block is assigned and in this case the PPA for the host write becomes non-contiguous. Therefore, one host write request requires two L2P entries. HP-UFS utilizes all response types of request commands to piggyback L2P entries. However, it is not guaranteed to transfer all L2P entries in time to the host HP-UFS driver. Consequently, the L2P entry transfer can be backlogged.

To this end, HP-UFS introduces *multi L2P entry transfer command*, a new command that fetches multiple L2P entries at once. In our prototype, this command fetches new L2P entries in batch. When the storage firmware experiences back logging in its L2P log buffer, it sets a hint for the host driver to issue the multi L2P entry transfer command. This approach helps improve the speed of L2P transfer from the storage device to the host driver.

C. Handling Host Read Request

HP-UFS handles the host read request by utilizing the cached L2P mappings in the L2P segment cache (① in Figure 4). When an L2P entry for a desired logical block is cached in the host memory, its PPA is used to issue an HPB

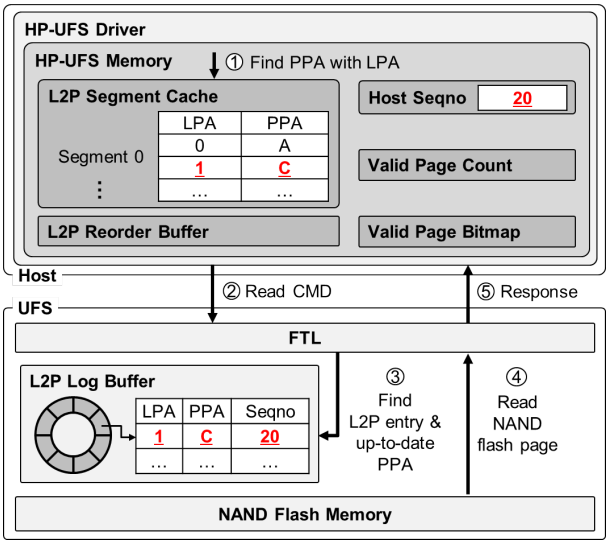


Fig. 4. The sequences of host read request in HP-UFS.

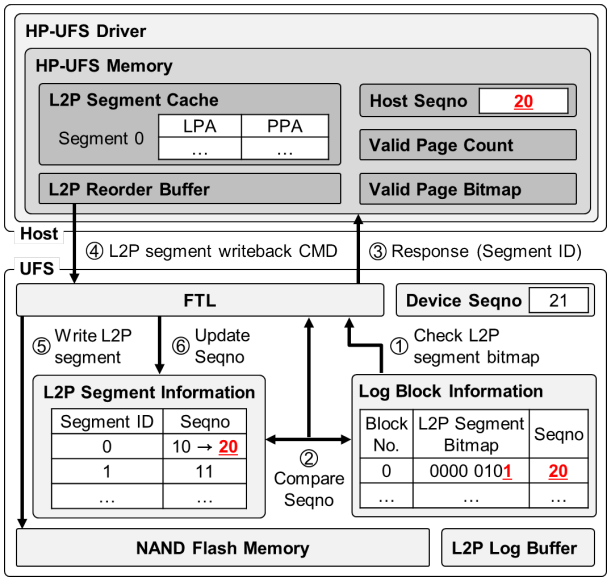


Fig. 5. The sequences of L2P segment writeback in HP-UFS.

read I/O command to the storage device, which is identical to the previous work [5], [12]. In the case of an L2P segment miss, the host driver initiates an L2P segment read command to fetch a missing L2P segment. After reading the segment, the driver proceeds with the host read I/O request.

When the HP-UFS storage device receives a read command with piggybacked PPA found in L2P segment cache (②), it checks whether the PPA is up-to-date. To achieve this, in HP-UFS, every read command also contains the host sequence number, which identifies the latest L2P entry the host driver has processed (Section III-B). The host sequence number identifies the age of the host L2P segment cache. Thus, when the HP-UFS storage controller processes a read request, it compares the host sequence number with those of L2P entries in the L2P log buffer (③). If an L2P entry of identical LPA of the read command is found, the controller replaces the PPA with the latest one from the L2P entry and then processes the read command (④), finally transmitting the data to the host (⑤). Consequently, even if an L2P entry propagation is backlogged, the storage device can process the read command with the correct PPA. It is important to note that this scanning operation takes place in SRAM L2P log buffer, which is much faster than the L2P segment invalidation followed by (slow) L2P segment read in the previous work (i.e., HPB [5]).

Another usage of the host sequence number supplied during read command processing is to determine which L2P entries in the L2P log buffer can be safely discarded. The firmware can identify the L2P entries in the log buffer that have been updated in the host L2P segment cache. These entries can then be safely removed from the L2P log buffer.

D. L2P Segment Writeback

In HP-UFS, new L2P mappings are stored in the host L2P mapping cache, which is volatile DRAM. Whenever an L2P mapping is changed, its L2P segment should be eventually written in flash memory. HP-UFS reduces the frequency of writing L2P segments since it can buffer plenty of new L2P mappings in the host DRAM. This characteristic is different from that of UFS or HPB where whenever the internal SRAM is full of new L2P mappings, they need to update L2P segments by performing read-modify-write of L2P segments. Hence, HP-UFS performs fewer L2P segment writes than UFS or HPB. Nevertheless, it is necessary to write new L2P mappings in the flash memory for the persistence of L2P mappings.

HP-UFS supports the writeback of dirty L2P segments by introducing an *L2P segment writeback command*. An L2P segment is considered dirty when it contains new L2P mappings that are modified from those in the flash memory. Dirty L2P segments are written back to flash memory in two ways: when the L2P segment cache in the host memory shrinks due to memory shortage [4], [5] or when the FTL transfers a hint of dirty L2P segment writeback to the HP-UFS host driver. Either way, the L2P segment writeback command is used; one command writes one L2P segment in the flash memory. The actual operation is similar to the host write request. The command is accompanied by the host sequence number to indicate the age of this L2P segment. When the L2P segment is written on the flash memory media, its host sequence number is also stored and is cached in SRAM. Storing the host sequence number with an L2P segment is important to avoid writing of stale L2P mappings.

In a normal scenario, the FTL firmware initiates the write-back of dirty L2P segments. Figure 5 illustrates this operation in detail. HP-UFS identifies flash blocks whose relevant L2P mappings are not persisted in flash memory. This can be done by recording flash blocks that have hosted new flash page writes recently. These flash blocks are called *log blocks*. Each log block has two fields, *L2P segment bitmap* that describes L2P segments relevant to the flash pages in this log block, and *log block sequence number* that identifies the latest L2P mapping entry this block contains.

The device checks an L2P segment bitmap per log block to identify L2P segments that need to be written to flash memory (① in the figure). This check starts from the oldest log block and examines the L2P segment bitmap to find the L2P segments that require writing. The device firmware checks whether the L2P segment on the flash memory is newer than the log block by comparing their sequence numbers (②). If the L2P segment on the flash memory is newer than the log block, the firmware skips this L2P segment. Otherwise, the firmware sends the id of the L2P segment to the host driver (③) to make the host perform the *L2P segment writeback command*. This is the case presented as the example in Figure 5. Log block 0 has the sequence number of 20 which is newer than the sequence number of L2P segment 0. Since L2P segment 0 is set in the bitmap of log block 0, the FTL firmware asks the host driver to perform the L2P segment writeback. When the host driver issues an L2P segment writeback command, the L2P segment is augmented with the host sequence number to indicate its age (④). The device writes the received L2P segment to the NAND flash memory (⑤) and updates the sequence number of the L2P segment with the received host sequence number (⑥); the sequence number of L2P segment 0 is changed from 10 to 20 in the figure. This update ensures that the sequence number of the L2P segment is synchronized with the host and reflects the latest modification.

The L2P segment writeback is also triggered when the host L2P segment cache needs to shrink [4], [5]. There may exist situations where the host is under memory pressure. In this case, the victim L2P segment is selected based on the memory management policy [4], [5]. Then, the L2P segment writeback command is issued if the victim segment is dirty, hence having up-to-date L2P mappings that are not written on the NAND flash memory. After the writeback, the victim L2P segment is discarded to secure free memory. Notably, this writeback operation skips processes ①, ②, and ③ in the figure.

E. GC and Wear Leveling

In HP-UFS, we address this issue by managing valid page bitmaps and valid page count in the host memory. When the device needs to trigger GC, it requests the necessary information from the HP-UFS host driver, allowing for more efficient management of valid pages and avoiding unnecessary flash memory accesses.

Managing the two data structures correctly is a challenging issue. In HP-UFS, whenever there is a change in an L2P mapping, the relevant information is encoded in an L2P entry and transferred to the host driver. Upon processing a new L2P entry in the host driver, the valid page bitmap and valid page count are updated prior to reflecting the new L2P entry in the cached L2P segment. The change of an L2P mapping involves moving an LPA from a previous PPA to a new PPA. The previous PPA, which is already stored in the cached L2P segment, is used to clear the corresponding bit of the valid page bitmap and decrease the valid page count. After that, the new PPA in the L2P entry is used to update both data structures. When the host driver is missing a cached L2P segment, the new L2P entry is temporarily staged in the reorder buffer. Once the cached L2P segment is retrieved from the device, the host driver performs the necessary operations to update the data structures.

When the device firmware runs out of free blocks, it triggers GC. First, the device embeds a hint within the response. The hint is to request the valid page bitmap of a block with the lowest valid page count. When the host driver receives such a hint, it sends a block number with its valid page bitmap. When the device firmware receives them, the device performs GC. Meanwhile, for wear-leveling the device firmware sends the hint in a response with a particular block number. This is because the device internally manages the program-erase cycle of each block so as to be able to pick a flash block for wear-leveling. In this case, the host driver sends the valid page bitmap of the specified block. Then, the two operations identify valid pages and copy them to a new flash block (i.e., log block) using the received valid page bitmap.

An important issue when the device uses the valid page bitmap received from the host is that the bitmap may not be accurate due to concurrent page write handling in the storage firmware. This can affect the device performing GC on invalid pages by potentially forwarding incorrect L2P entries to the host driver. In general, when concurrent host writes occur for the same LPA, the sequence number of each L2P entry can be used to identify the order of the operations. When a host write and GC occur almost concurrently for the same LPA, if host write occurs after GC, this situation has no problem because the sequence number of the host write is newer than that of GC. However, if GC occurs after host write, this situation can be misinterpreted as the latest logical page is at the PPA GC has made. This is incorrect since the latest logical page is at the PPA the host write has made. This problem happens when GC has copied an invalid page whose invalidity is not transferred to the host driver on time.

To address this problem, HP-UFS includes the source block number field in the L2P entry. This field is set when the L2P mapping is changed by GC or wear-leveling. When the host receives and processes such L2P entries, the host driver checks the source block number and verifies whether the source block number matches the block number of the current cached L2P segment. If they match, the L2P entry is reflected in the L2P segment cache. Otherwise, the L2P entry is discarded; the L2P change of the host write is already reflected in the L2P segment and this PPA is the correct one. Hence, even though the sequence number of the GC-induced L2P entry is later than the host-write-induced L2P entry, the newer one is discarded not to override the L2P mapping host write has made.

F. L2P Segment Cache Management

Modern mobile systems, such as smartphones, are equipped with a substantial amount of DRAM. For instance, flagship smartphones come with more than 12 GB of memory [42], and mid-range phones feature 4–8 GB of memory [43]. Although these devices have a large amount of installed DRAM, borrowing and utilizing host memory should be done

with careful consideration of its minimal impact on mobile system workloads [4], [5].

In mobile systems, a large portion of memory is not used by foreground applications, but rather for caching background applications [44]–[46]. In this regard, borrowing a small additional amount of memory may not significantly impact the performance of foreground applications. The amount of memory required to accelerate mobile storage I/O is relatively small, ranging from a few to tens of MBs. This is due to the fact that the typical I/O footprint of mobile workloads is on the order of up to a few GBs [47], necessitating only a few MBs for L2P mappings. Consequently, borrowing and utilizing up to a few tens of MBs of DRAM is unlikely to cause immediate performance degradation in foreground applications. Thus, we believe that using a small amount of host memory to enhance mobile storage I/O is more beneficial than allocating it for caching background applications.

In addition, various techniques can be augmented to improve the memory efficiency of HP-UFS. Basically, in HP-UFS, the size of the L2P segment cache in the host memory is configurable and the least-recently-used (LRU) replacement is applied. Hence, when the L2P segment cache is full, the least recently used L2P segment is evicted to secure space. Moreover, timer-based eviction [5], [13] can also be applied, which evicts unnecessary L2P segments eagerly to secure more space. Finally, application or system-level hints can be exploited for efficient L2P segment cache management as well as the foreground application performance [4]. These approaches are complementary to HP-UFS since the main goal of HP-UFS is accelerating both read and write I/Os by utilizing the host memory; in our experiment, only the LRU replacement is applied.

G. Crash Consistency of L2P Mappings

It is important to provide the crash consistency of L2P mappings. HP-UFS stores new L2P mappings in volatile memory as much as possible; a new L2P mapping is transferred from the storage device to the host memory and is written back when the L2P segment writeback is performed. Keeping new L2P mappings in volatile memory is not the sole problem of HP-UFS but the problem of UFS as well. In UFS, new L2P mappings are buffered in SRAM. Hence, when a sudden power failure happens, these new L2P mappings are lost. Therefore, it is necessary to prepare a method to recover lost L2P mappings.

UFS recovers these lost L2P mappings by recording log blocks and replaying L2P mapping changes made by log blocks. As described in Section III-D, log blocks are flash blocks that have recently hosted flash writes. Hence pages in log blocks have their new L2P mappings. The UFS FTL records log blocks in a meta block in flash memory. Therefore, when a sudden power failure occurs, log block information in a meta block can be used to identify log blocks and replay L2P mapping modifications in log blocks [48], [49]. Please note that a flash page has its LPA in its spare area; hence necessary information for replaying is stored in the flash memory.

TABLE I
FLASH MEMORY CONFIGURATION.

	SLC [51]	**TLC [52]**
Page read latency	25 usec	60 usec
Page program latency	150 usec	550 usec
Page: 16 KB, Pages per block: 64 Blocks per chip: 8 K, Chips: 8, Plane: 2		
Controller SRAM size: 1.5 MB, Over-provisioning: 15%		

TABLE II
HINTS EMBEDDED IN I/O RESPONSE IN HP-UFS.

Type field in I/O response (Device → Host)	
L2P segment write request	Section III-D
Multi L2P entry transfer request	Section III-E
Valid page bitmap transfer request	Section III-E

HP-UFS also utilizes this method of recovering L2P mappings. In HP-UFS, it is challenging to limit the number of log blocks since virtually HP-UFS has an unlimited amount of memory for buffering new L2P mappings (in the host DRAM). Instead, HP-UFS keeps the number of log blocks limited considering the recovery time. The UFS protocol specification defines a maximum duration for device initialization [2], [50]. Therefore, in order to adhere to this time limit, HP-UFS implements a restriction on the number of log blocks that have not yet flushed their L2P segments to the NAND flash memory. Once this limit is reached, the HP-UFS device firmware enforces the writeback of L2P segments from the oldest log block (Section III-D). When all the dirty L2P segments associated with a log block are written back on the flash memory, the log block becomes a normal block. This ensures that the recovery process can be completed within the specified time frame while not losing any L2P mappings. In our evaluation, we set the number of maximum log blocks to 8; we found this is small enough to bound the device recovery time assuming the worst case (see details in Section IV).

IV. EVALUATION

A. Methodology

As it is practically infeasible to modify the UFS firmware of commercial off-the-shelf SSDs, we developed the prototype of HP-UFS using FEMU [21]. The FEMU emulator is an accurate, scalable and widely-used flash emulator capable of emulating various types of flash-based storage systems; more than 34 papers [53] have used FEMU to prototype and evaluate their ideas. We carefully configured the low-level flash parameters to make it perform like a real UFS 3.0 device [22]. Our UFS emulator consists of eight parallel flash chips that can be accessed in parallel through two planes and eight channels. Single-level cell flash chips are used to store flash memory metadata, such as L2P segments. Triple-level cell [52] flash chips are used to store user data blocks. Table I shows the detailed parameters used in the emulator. The UFS emulator is configured to have a 1.5 MB SRAM. Furthermore, we modified FEMU to support demand loading-based FTL and HPB protocols to demonstrate the effectiveness of the HP-UFS. The SRAM is used as shared memory that is dynamically allocated to the write buffer, L2P segment buffer,

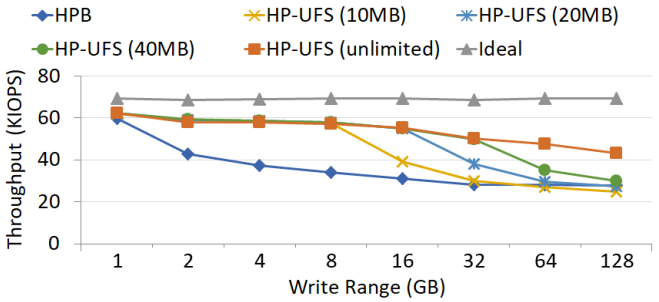


Fig. 6. Random write performance with varying write ranges (I/O unit: 4 KB, total I/O size: 1 GB, I/O depth: 8) (*unlimited* in the figure denotes a case wherein all L2P mappings for UFS are cached in the host DRAM).

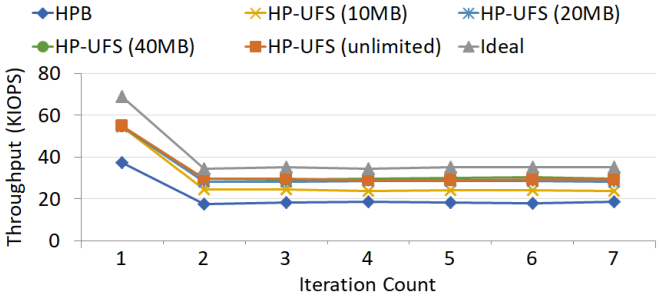


Fig. 7. Random write performance under GC (I/O unit: 4 KB, write range: 16 GB, total I/O size: 1 GB, I/O depth: 8).

GC buffer and so on. As a result, FTL repeatedly allocates and releases 4 KB of shared memory depending on the workload.

Since the FEMU emulator uses the non-volatile memory express (NVMe) protocol [54], we modified the submission and completion queues to accommodate the UFS/HPB/HP-UFS command/response sets. The command submission entry is modified to include 8 bytes of extra data as in HPB [5], [12]. HP-UFS uses 8 bytes to transfer PPA and host sequence number to the storage device. The completion queue entry includes an additional 16 bytes of fields for implementing HP-UFS. This matches the unused 16 bytes of the command response packet of the UFS Protocol Information Unit (UPIU) [2], [50], [55]. The extra fields in the completion entry contain 4 byte LPA, 4 byte PPA, 2 byte source block number, 1 byte length, and 1 byte type. The LPA, PPA, source block number and length fields are used to transfer an L2P entry to the host device driver. The type field is used to piggyback a hint that triggers the reaction by the HP-UFS device driver. The list of hints and the reaction of the device driver is summarized in Table II. In the HP-UFS host memory, the L2P reorder buffer has 2K 16 B entries, the valid page bitmap is 4 MB, the valid page count is 32 KB. The size of the L2P segment cache is configurable and the LRU replacement is applied [1].

We compared our scheme with the following schemes:

- **UFS** that is the baseline DRAM-less mobile flash storage.
- **HPB** that is the previous work that utilizes the host DRAM but accelerates only block reads [5].
- **Ideal** that is an ideal flash storage device with plenty of DRAM, hence showing no L2P mapping-related overheads.

B. Micro-benchmark Workload

We measured the basic I/O performance using the flexible I/O tester (FIO) microbenchmark suite [23].

Random write performance. First, we evaluated the write performance according to the size of the write range. Figure 6 shows the random write performance of each scheme, along with the performance patterns of HP-UFS depending on the cache size limit in host DRAM. HPB does not cache L2P mappings for writes. HP-UFS improves the write performance close to *Ideal* by leveraging DRAM to accelerate block writes, and it shows a throughput improvement of up to 77% compared to HPB when the write range is 16 GB. As the write range expands, our scheme results in an increased number of L2P segment writebacks, leading to a decrease in performance. Furthermore, when the write range approaches the reach of L2P cache (e.g., 20 MB cache can cover up to 20 GB of SSD I/O range), the performance begins to decrease. However, our scheme shows maximum efficiency as if there are no memory constraints (HP-UFS in the figure), even when the cache size is only 20 MB and the I/O range is 16 GB, which is large enough to cover typical I/O footprint of mobile workloads [47]. When considering a mobile device with approximately 4GB of DRAM [43], it suggests a performance enhancement for datasets up to 16 GB, achieved by borrowing only 0.5% of DRAM.

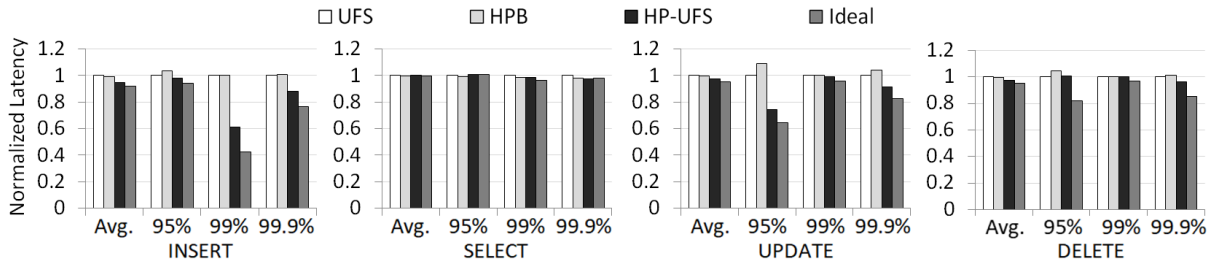
Garbage Collection (GC) performance. Next, we measured the I/O performance under GC. To create a situation where GC occurs, we performed a random write of 100 GB to the storage in advance. Then, we measured the performance for each 1 GB random write iteration and depicted the results in Figure 7. In the first iteration, no GC takes place, leading to maximum throughput for each scheme. The throughput has decreased and remains stable from the second iteration due to GC taking place in all schemes.

When GC occurs, HPB loads and retrieves L2P segments from NAND flash memory for searching valid pages. On the other hand, HP-UFS has no cost for searching valid pages since it retrieves the valid page bitmap from the host HP-UFS driver. Therefore, the throughput of HP-UFS outperforms that of HPB by up to 70%. Unfortunately, HP-UFS cannot reach the result of Ideal due to the interference caused by creating new L2P mapping entries with GC and performing L2P segment writeback operations to send them to the host.

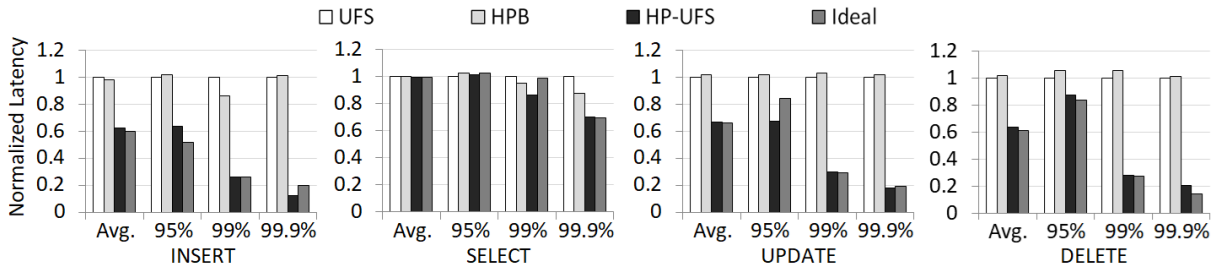
C. Real-world Workload

To evaluate the impact of HP-UFS on user experience, we tested the performance of HP-UFS with two real-world workloads: SQLite and mobile application usage. We use Mobibench [24] for evaluating the SQLite [56] workload. For the mobile application usage workload, we collect the traces of block I/O operations while using smartphones under various application use scenarios, and then replay the traces. We collected the I/O traces on a real smartphone, Google Pixel 4 [57] using the F2FS file system [18]. When we

[1]The source code is available at https://github.com/s3yonsei/hpufs_public.



(a) Without GC



(b) With GC

Fig. 8. Comparison of SQLite performance for each query operation (normalized to results in UFS).

collected the I/O traces, we aged the file system to mimic real storage usage [58]. The two real-world workloads, Mobibench and replaying block I/O traces of real application usage are performed on the flash storage device using the FEMU emulator. We evaluated the performance under two conditions, without GC and with GC. With GC, 97% of the FEMU storage device is occupied with dummy files to make the FTL perform GC. The L2P segment cache size set to 20 MB (assuming 0.5% of 4 GB DRAM) for HP-UFS and HPB. Please note that, the `fsync/fdatasync`-related I/O commands are faithfully implemented. Hence, when the *flush* command is issued, the FEMU emulator flushes its internal write buffer. The *FUA* command is also implemented so as to perform writes bypassing the write buffer. Please note that these commands do not cause the writeback of dirty L2P segments in UFS, HPB and HP-UFS since the L2P mappings are crash-consistent as explained in Section III-G.

SQLite performance. We evaluate SQLite performance by comparing the latency for each query operation within the SQLite database. Figure 8 shows both the average and tail latency. One thing to note is that we implement the *SELECT* query operation into the Mobibench SQLite benchmark to comprehensively measure the SQLite performance.

When there is no GC, the latency difference between HPB and HP-UFS is caused by write I/Os. Write I/Os cause two types of overheads in HPB, invalidation of cached L2P segment and the write-back of L2P segment to flash memory due to mapping changes. However, in HP-UFS, the two types of overheads are effectively eliminated due to the propagation of L2P mapping changes from the device to the host memory and the delayed writeback of L2P segments. Therefore, the performance improvement is observed on the workloads with writes, i.e., *INSERT*, *UPDATE* and *DELETE*. Our scheme shows significant performance improvement, especially in tail latency, reducing up to 39% in the 99.9th percentile latency for *INSERT* queries when compared to HPB.

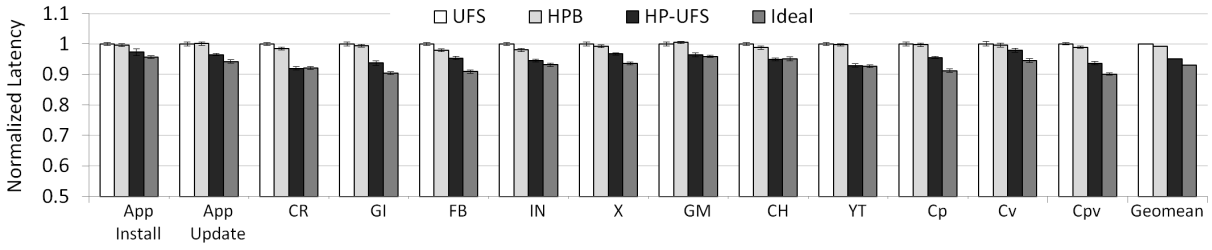
TABLE III
APPLICATIONS (ABBREVIATION) AND USE SCENARIOS FOR EACH CASE.

Application	Use Scenarios	Read/Write (MB)
Clash Royale (CR)	Play a stage	7 / 90
Genshin Impact (GI)	Play a quest	168 / 11
Facebook (FB)	Browse & read posts	0.8 / 180
Instagram (IN)	Browse & read posts	3 / 236
X (X)	Browse & read posts	14 / 83
Google Maps (GM)	Find & navigate place	27 / 93
Chrome (CH)	Browse multiple tabs	38 / 26
Youtube (YT)	Watch videos	5 / 111
Camera-picture (Cp)	Take pictures	2 / 86
Camera-video (Cv)	Record a video	13 / 1729
Camera-play-video (Cpv)	Play a video	1782 / 17

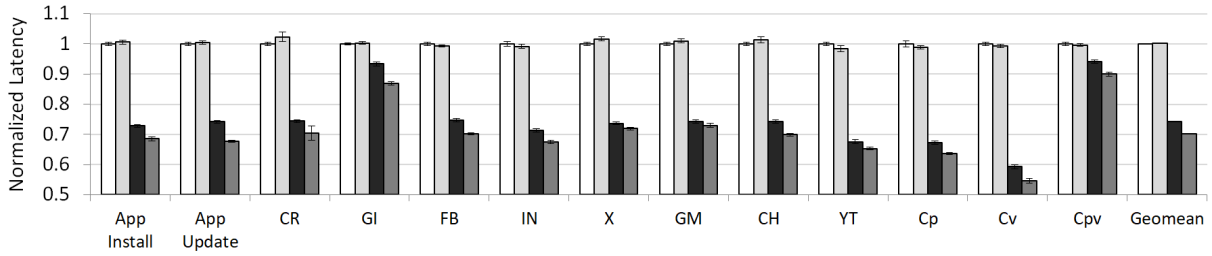
In experiments where GC occurs, HP-UFS notably reduces query latency compared to previous schemes. As mentioned in Section III-E our scheme manages the valid page bitmap in host memory, leading to no additional cost for searching valid pages during GC. In comparison to HPB, we observe a decrease of up to 37% in average latency in *INSERT* queries, and up to 88% in tail latency in the same queries.

Application performance. We collected the I/O patterns of several application use scenarios on a Google Pixel 4 device mounted with an F2FS file system and replayed them on the emulator. Table III shows the applications employed in our evaluation and their respective use scenarios. Additionally, we composed *AppInstall* workload for sequentially installing applications from *CR* to *YT* in the table, and *AppUpdate* workload for sequentially updating them after installation.

We compare the performance across different schemes for each workload, as shown in Figure 9. Without GC inside the device (Figure 9(a)), HP-UFS improves the performance



(a) Without GC



(b) With GC

Fig. 9. Comparison of real-world application performance (normalized to results in UFS).

of real-world applications by up to 7% in *YT*, and 4% in geometric mean compared to HPB. This modest performance improvement is attributed to the small write range despite most of these workloads being write-intensive. Figure 10(a) shows the breakdown of the overheads incurred at each scheme in the same workload as Figure 9(a), especially for representative read or write-intensive workloads. In comparison to HPB, HP-UFS demonstrates a reduction in both L2P segment write overhead and L2P segment read overhead. These findings are consistent across the remaining workloads.

Furthermore, when GC occurs during the workload (Figure 9(b)), the performance of HP-UFS exhibits a significant improvement over HPB. Compared to HPB, HP-UFS shows performance improvements by up to 40% in *Cv*, and 26% in geometric mean. During GC, there is not only the cost for searching valid pages (*Valid Page Search* in Figure 10(b)) but also an increase in the L2P segment write for the updated L2P segment, which affects the overall performance of the workload. As shown in the figure, HPB shows about 21-27% of valid page search overhead for GC among the write-intensive workloads. However, our HP-UFS eliminates that overhead by managing valid page bitmaps in host memory. Moreover, the L2P segment write overhead, which accounted for about 4-13% on HPB, decreased to about 1-6% on HP-UFS. We verified that HP-UFS effectively reduced L2P segment write overhead by observing 37-80% lower latency compared to HPB. In the context of read-intensive workloads (e.g., *GI*, *Cpv* in Table III), the removal of valid page search overhead has had a minimal impact on overall latency, resulting in only modest performance improvements as shown in Figure 9(b).

D. L2P Mapping Table Recovery Overhead

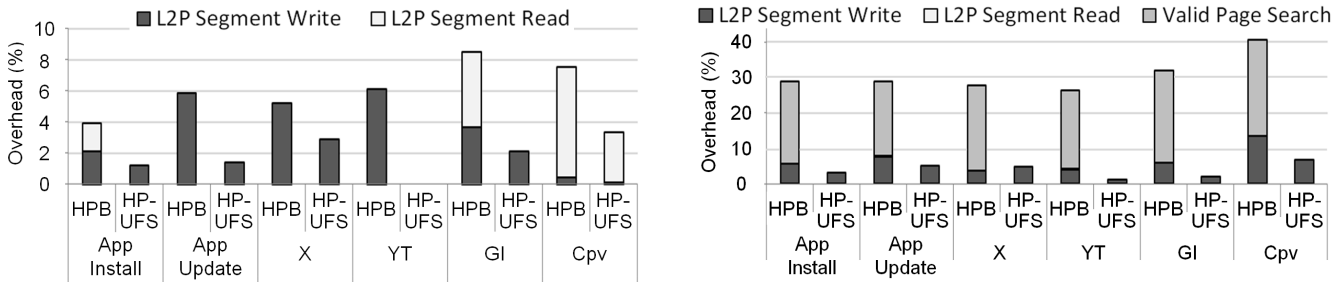
When the L2P mapping table is recovered during boot time due to unexpected situations (e.g., power failure), two major factors affect the overall recovery latency: the number of recovered L2P segments and the number of data page scan counts required for the recovery process. These factors can be larger on our HP-UFS than HPB because our scheme caches L2P entries in host memory, which may cause L2P segment writes to be delayed. As a result, the number of L2P segments to be recovered may increase, and the overall recovery time may also increase.

UFS devices must complete the recovery process within the specified time limit, with a maximum device boot time of 1.5 seconds [2], [50]. Our evaluation demonstrates that our system does not exceed this limit even under worst-case recovery conditions. Figure 11 shows the latency for the recovery process according to the limit value of the two factors. The number of recovered L2P segments significantly impacts overall recovery latency more than the number of data page scan counts. This is because L2P segments require both read and write operations for recovery, while data page scans involve only read operations. In our evaluation, the maximum number of L2P segments that can have is 32,768; the total 8 log blocks (each 16 MB block) can make 32,768 distinct L2P entries (each 4 KB page) each of which belongs to a distinct L2P segment. Even when the number of recovered L2P segments reaches the maximum (32,768 L2P segments), which is the worst-case recovery condition in our experimental setup, the recovery process takes about 300 ms.

Our evaluation emphasizes the importance of managing and minimizing the number of L2P segments involved in the recovery process to achieve efficient recovery and meet the specified boot time limit. While reducing the number of data page scans is also beneficial, it may have a relatively smaller impact on the overall recovery latency.

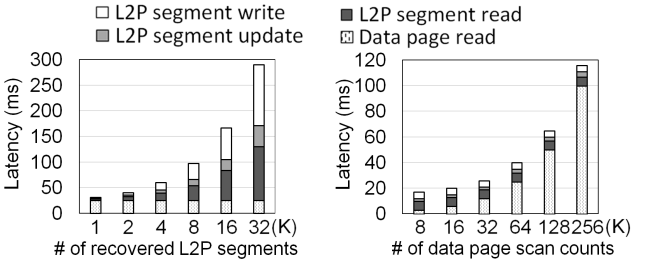
V. Related Work

Caching Mapping Tables in Host Memory. Recent studies have explored caching mapping tables in host memory to



(a) Without GC (b) With GC

Fig. 10. A breakdown of the latency overhead for real-world workloads (each overhead represents the time taken by each operation within the total I/O time in NAND flash memory.).



(a) Varying # of recovered L2P segments (b) Varying # of data page scan counts

Fig. 11. Latency breakdown of L2P mapping table recovery process (fixed 32,768 data page scan counts for (a) and fixed 4,096 recovered L2P segments for (b)).

enhance the storage performance in various storage protocols, including SCSI [59], HMB with NVMe [6], [60], and HPB with UFS [12]. Motivated by the scarcity (or absence) of DRAM within a storage device, they concentrated solely on enhancing I/O performance using host resources, regardless of whether both read and write operations were involved. Among them, HPB [5] is the first commercialized work based on UFS. HPB simplifies the host system architecture by only requiring the management of mapping tables on the host side without requiring additional hardware design. HPBvalve [4] focuses on improving read I/O performance by prioritizing and managing the mapping address table entries in the HPB memory around foreground applications. However, despite the significant impact of write, GC and wear-leveling performance in mobile environments, there has been relatively little attention given to leveraging host memory to improve the performance of these operations.

HP-UFS takes a step further in improving mobile flash storage performance. It enhances write performance by reducing the number of writes in the mapping tables, encompassing both sequential and random writes, through the utilization of host memory. Additionally, HP-UFS proposes an efficient management technique for the valid page bitmap in host memory, minimizing the cost of valid page search overhead and thereby improving GC and wear-leveling performance. By leveraging host memory effectively, HP-UFS optimizes read, write, GC and wear-leveling operations, offering a comprehensive solution for mobile flash storage systems.

Small-Footprint Mapping Tables. One possible solution to mitigate the limited internal SRAM capacity of mobile storage is to reduce the size of the mapping table. Block-level mappings [25] are a plausible design choice for reducing the mapping table sizes; however, they achieve poor performance due to their high write amplification during random writes and GC. As a result, page-level mappings [9], [26], [61]–[63], which incur larger mapping tables with lower write amplification, have become the de-facto for flash storage. Aimed at further reducing the mapping table sizes, prior studies utilized machine learning techniques to overcome the space-consuming one-to-one L2P mappings. For example, LeaFTL [32] leveraged linear regression to replace the one-to-one mappings with learned index segments. Unfortunately, the size of the learned index segments still remains DRAM-scale and cannot fit in SRAM, which hinders their adoption in mobile storage.

VI. Conclusion

In this paper, HP-UFS is proposed to show almost full performance potential of flash-based mobile storage devices. HP-UFS overcomes the limitation of DRAM-lessness of mobile storage systems by borrowing and utilizing the host DRAM. While prior work like HPB has utilized host DRAM to improve the read performance, HP-UFS further accelerates all the storage operations, such as write, GC and wear-leveling. This can be achieved since the host device driver and the FTL firmware work in harmony to exchange necessary data while performing flash memory writes followed by the change of L2P mappings. Through extensive evaluation of our thoroughly implemented HP-UFS prototype, we demonstrate that HP-UFS outperforms both UFS and HPB in micro-benchmark workloads and real-world workloads.

VII. Acknowledgments

We would like to thank the anonymous reviewers for their valuable comments. This work was supported partly by Institute for Information & communications Technology Promotion (IITP) grant funded by the Korea government (MSIP) (No.2021-000773, Research on Edge-Native Operating Systems for Edge Micro-Data-Centers), and by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (RS-2023-00321688) and by Samsung Electronics.