Improving Flash Storage Performance by Caching Address Mapping Table in Host Memory

Wookhan Jeong, Yongmyung Lee, Hyunsoo Cho, Jaegyu Lee,
Songho Yoon, Jooyoung Hwang, and Donggi Lee
S/W Development Team, Memory Business, Samsung Electronics Co., Ltd.

Abstract

NAND flash memory based storage devices use Flash Translation Layer (FTL) to translate logical addresses of I/O requests to corresponding flash memory addresses. Mobile storage devices typically have RAM with constrained size, thus lack in memory to keep the whole mapping table. Therefore, mapping tables are partially retrieved from NAND flash on demand, causing random-read performance degradation.

In order to improve random read performance, we propose HPB (Host Performance Booster) which uses host system memory as a cache for FTL mapping table. By using HPB, FTL data can be read from host memory faster than from NAND flash memory. We define transactional protocols between host device driver and storage device to manage the host side mapping cache. We implement HPB on Galaxy S7 smartphone with UFS device. HPB is shown to have a performance improvement of 58 - 67% for random read workload.

1. Introduction

Flash memory based storage devices are widely used in mobile devices, desktop PCs, enterprise servers, and data centers. Flash memory has operational limitations such as erase-before-write and sequential write on memory blocks. Flash Translation Layer (FTL) software manages such constraints and abstracts flash memory chips as a block device.

FTL maintains its mapping table to perform address translation from logical addresses (block numbers) to physical addresses (flash memory block and page numbers). FTL mapping methods can be classified into three types [8]: block mapping, page mapping, and hybrid mapping. The size of mapping table depends on the type of mapping scheme. Since block mapping [1] scheme provides mapping at the granularity of flash memory's block, it needs a relatively small mapping table. However, its random write performance is poor, because a large number of pages in a block are copied to a new memory block to update a page in the memory block. In contrast, page mapping [2] appends every updated page in a log block and maintains page granularity mapping. Hybrid mapping is an extension of block mapping to improve random writes [3, 4, 5, 6]. It keeps a smaller mapping table than page mapping while its performance can be competitive to that of page mapping for workloads with substantial access locality.

Among the three types of mapping, page mapping performs most efficiently, but it requires a large mapping table (typically 1/1024 of device size when using 4Bytes mapping entry for 4KiB sized page). If the whole mapping table cannot fit into device's memory, FTL loads a part of the table on demand [7], which may increase latency and hence degrade the performance of random read I/O.

Random read performance has become more important as computing power is enhanced with the rise of rich OS features (e.g. multi-window in Android N) and applications (MSFT office packages for mobile). However, flash storage for mobile devices still suffer from limited memory resources due to its constraints on power consumption and form factors. Properly managing the size of mapping table is a serious concern as device density increases.

In this paper, we propose Host Performance Booster (HPB), a new method of caching mapping table to host memory without additional hardware support in mobile storage devices, to address the mapping table size problem. We show design and implementation of HPB on a smartphone with UFS (Universal Flash Storage) which is a new generation mobile storage using SCSI command set. We describe the transaction protocol to exchange mapping table information between device driver and UFS device. Experimental results show that random read performance is improved up to 67% by HPB.

Section 2 describes the background of FTL operations and overviews HPB. Section 3 explains the details of HPB implementation. We show experimental results and related works in Section 4 and Section 5, respectively, and conclude in Section 6.

2. Design Overview

2.1. FTL operation background

FTL maintains Logical-to-Physical (L2P) mapping table in NAND flash memory. In general, flash memory devices used for mobile systems do not have DRAM, but

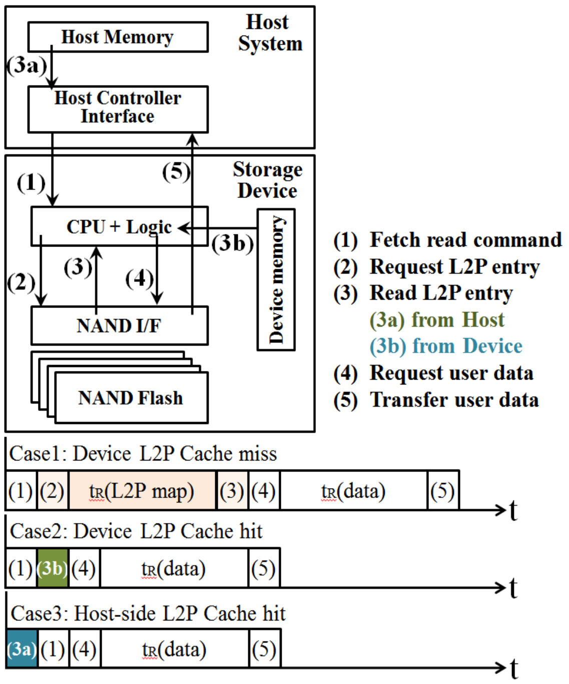

Figure 1: Legacy mobile storage operation as per read request. tR means the latency of reading a flash memory page.

SRAM of limited size in storage controller, where FTL caches recently used L2P mapping entries. On receiving a read request, FTL looks up the map cache to retrieve a corresponding mapping entry, or loads the mapping from flash memory in case of cache miss. Cache miss penalty is significant for small chunk random reads. For example, to process a 4KB size read request, two read operations (one for map and one for data) are required in case of L2P cache miss.

The overall procedure of processing a read request is depicted in Figure 1. Device fetches a read request, looks up its L2P cache in device SRAM. On L2P cache miss (Case1), L2P in flash memory is loaded, which takes hundreds of microseconds. On cache hit (Case2), L2P entry is retrieved from device SRAM.

2.2. HPB Overview

The key idea of HPB is that host device driver caches L2P mapping in host memory (DRAM) and sends the corresponding L2P information piggybacked in an I/O request to device whether the L2P entry is cached in host memory. Since the L2P information is provided by the host in the request, device does not have to load L2P entry from flash memory even on its internal L2P cache miss (Case3 in Figure 1). Device investigates the host provided L2P to protect data against possible corruption of L2P metadata. In the following, we describe the HPB in more detail.

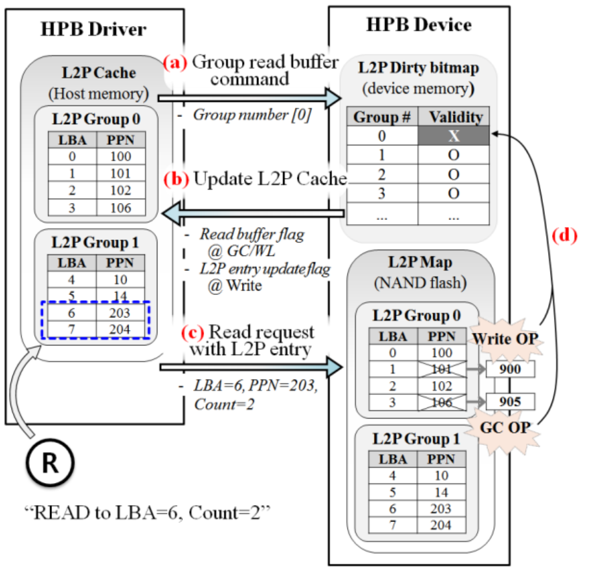

Figure 2: HPB related host-device interface

2.2.1. L2P Cache Initialization

At boot time, HPB device driver allocates kernel memory for L2P cache, requests L2P information to device, and populates the L2P cache.

2.2.2. Device to Host L2P information delivery

In HPB, L2P information is delivered from device to host driver in two ways. First, host driver can fetch a chunk of L2P entries by sending a command, implemented via SCSI READ_BUFFER command [11] (Figure 2(a)). This command can be issued only when the storage device is idle to avoid any impact on normal I/O performance. Second, device piggybacks L2P information in response packets of normal I/O requests (Figure 2(b)). While this does not incur command overhead, the size of information that can be contained in a response packet is limited. Both methods are used to synchronize the host side L2P cache with device's L2P mapping table, which is explained more in 2.2.4.

2.2.3. Host to Device L2P information delivery

HPB driver includes L2P information in an I/O request if host side cache has a corresponding L2P entry (Figure 2(c)). On receiving a read request with L2P information, device verifies if the given L2P has been published by itself, and checks whether that entry is up-to-date. If the given L2P passes those inspections, device uses it without loading the entry from flash memory. This will be described in more detail in 2.2.5.

2.2.4. L2P Cache Consistency

L2P mapping entries in device are updated not only by host's writes but also by FTL's internal operations such as wear leveling (WL), which is required to avoid block wear-outs, or garbage collection (GC), which is required to reclaim free space. HPB device notifies host of L2P change events by raising a flag in a response packet, then HPB driver fetches up-to-date L2P entries from device.

As shown in Figure 2(d), device maintains a bitmap to keep track of address ranges that have L2P change(s). Since this bitmap should reside in device memory, small bitmap is preferred. To reduce the bitmap size, we have a single bit which represents a group of consecutive pages. Device notifies host of which group has updated L2P entries, and then the host driver issues a READ_BUFFER command to fetch L2P entries for the group.

HPB device checks if given L2P is up-to-date by referring to the dirty bitmap, and ignores it if the group, to which the requested page belongs, is dirty. Since the group of page is large, a small change to the group may deteriorate performance. To keep the host side L2P cache consistent as early as possible, HPB device returns updated L2P entries in a response packet for normal I/O. Most of flash devices have an internal write buffer, so the requested data may not have been persisted when a write I/O request is completed. Therefore, it should be noted that the L2P information for a write request may be delivered in response packet of other requests later.

2.2.5. L2P information verification

As previously mentioned, since L2P map is critical to data integrity, HPB device verifies the L2P information given by the host before using it. Host side L2P cache may be modified by malicious software or kernel defects. HPB driver checks if the L2P entry has been published by itself. To do so, HPB device makes a signature using LBA and PPN, encrypts L2P entry using random seed and the signature as shown in Figure 3. Since encryption key is not provided to host, host cannot decrypt it. HPB device can detect whether the L2P entry has been tampered while in host memory by decrypting the entry and checking the signature. We do not implement the verification scheme in this work because the device used in our experiment does not have an encryption/decryption hardware engine.

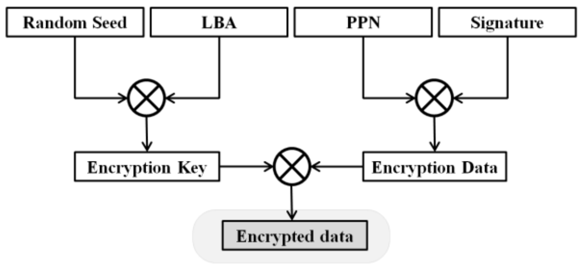

Figure 3: Encryption of an L2P entry

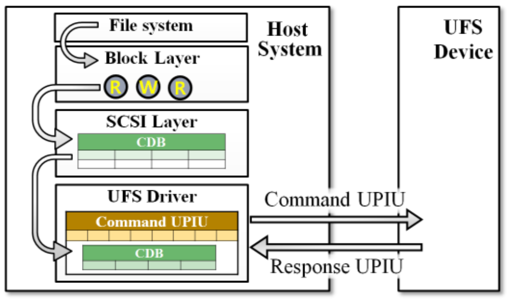

Figure 4: UFS protocol overview

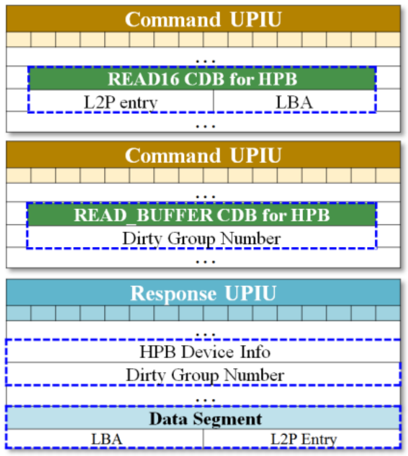

Figure 5: UPIU examples in HPB

3. Implementation

This section describes our implementation of HPB on a Galaxy S7 [12] smartphone with an UFS device [13] in order to prove the concept of HPB.

3.1. Host-Device Protocol on UFS

Here we describe the interface between HPB driver and HPB device for HPB operation.

Figure 4 shows how a Linux host and UFS device communicate. UFS storage stack is a layered architecture consisting of application layer and transport layer. CDB (Command Descriptor Block) and UPIU (UFS Protocol Information Unit) are the main structures in the application layer and the transport layer, respectively. CDB is a structure for a client to send a command to a SCSI device. UPIU is a data structure used to transfer information between UFS host and device. CDB is included in a "command" UPIU. Data is exchanged within "data" UPIUs. We explain our implementation using CDB and UPIU in the following.

1) As shown in Figure 5, we redefine the 8 byte LBA field of READ16 command's CDB in order to contain 4 byte LBA and 4 byte L2P entry.

2) UFS driver issues a READ_BUFFER command, containing a group number as shown in Figure 5, to

fetch L2P entries of the group from device. A group consists of 4096 L2P entries in our implementation. On receiving the READ_BUFFER command, UFS device responds with L2P entries of the group.

3) UFS device notifies host of dirty group using a flag in device information field of a response UPIU. If the flag is clear, for example, dirty group number is contained in the response UPIU. Then, HPB driver issues a READ_BUFFER command for the dirty group. On the other hand, if the flag is set, the latest L2P entry is included in Data segment in Response UPIU. Figure 5 shows CDB and UPIU fields mentioned above.

3.2. Device side L2P management

HPB device maintains a L2P dirty bitmap in its memory to manage L2P cache consistency to keep track of the dirty page groups. The whole LBA range is divided into several groups and one bit is assigned to each group. The size of group affects the overall performance of HPB because if there is any L2P change to a page in a group, device ignores L2P information of all pages belonging to the group. In our implementation, a group consists of 4096 pages.

HPB device marks a bit of the L2P dirty bitmap for a group when it processes host writes, GC, and WL operations. It clears the bit when host HPB driver fetches the latest L2P information via READ_BUFFER command or response UPIU.

3.3. Device Driver

One of our major design goals for HPB driver is to be transparent to the upper layer, including file system. HPB driver initializes during the initialization phase of UFS driver. It allocates kernel virtual memory for L2P cache and data structures to manage the cache. HPB driver performs the following three operations.

1) If UFS device sends L2P map entries in response UPIU, HPB driver writes these entries to its L2P cache.

2) If UFS device sends a dirty L2P group number in response UPIU, HPB driver issues the READ_BUFFER request with the group number, which is executed during device idle time to avoid impact on normal I/O. HPB driver updates its L2P cache using the data delivered as per the request.

3) When the upper layer requests a read to the UFS driver, HPB driver finds a corresponding physical address with its logical address and issues the read command which includes the physical address using READ16 command format.

3.4 Multi-page Read Request Handling

The implementation in this paper has a limitation that only one L2P entry can be transmitted in to a READ16 CDB. With this constraint, if the physical chunk corresponding to the logical address of the request is physically contiguous, the PPN information of the entire chunk of the request is transmitted with only one PPN transmission. Transmitting additional information can be supported by using extra header segment, which is under discussion in JEDEC for possible inclusion in the next UFS 3.0 standard. If extra header is supported, we do not need to change READ16 CDB fields as well.

4. Performance Results

On Galaxy S7 with UFS, we performed a series of micro benchmarks to analyze the HPB performance.

Random read performance: HPB performance benefit is expected to be more significant under smaller chunk random read requests, because the workload accesses L2P map frequently. To see the benefit for small random read requests, we use tiobench [14] that measures I/O performance using multiple threads. We configure tiobench to set 4KB record size on 1GB address range and use 1, 2, 4, 8 and 16 threads. As shown in Figure 6, HPB shows 59 ~ 67% of random read performance improvement, as the cache hit rate is 100% in the Host-side L2P Cache.

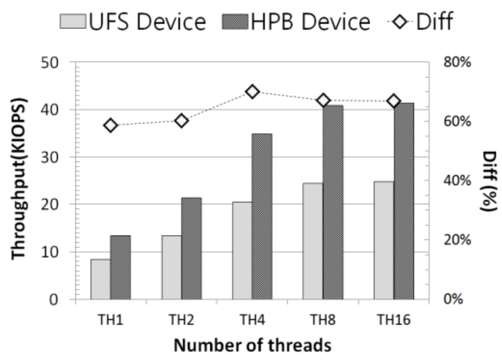

Figure 6: tiobench 4KB RR (Random Read) performance

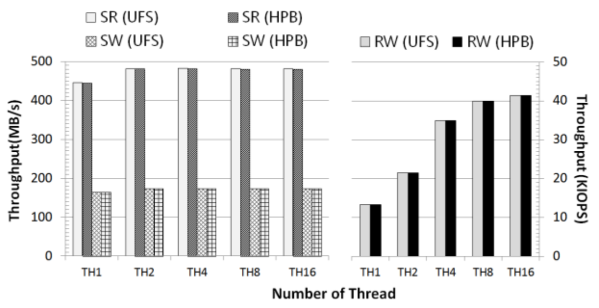

Figure 7: tiobench SR(Sequential Read), SW(Sequential Write), RW(Random Write) performance.

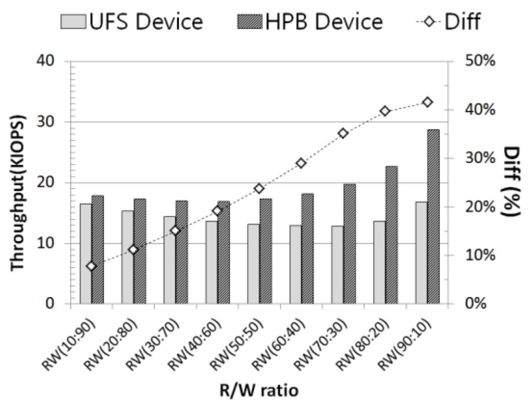

Figure 8: Mixed pattern performance (4KB record size, 1GB I/O issue, 16 threads). In RW(x:y), x is read portion and y is write portion.

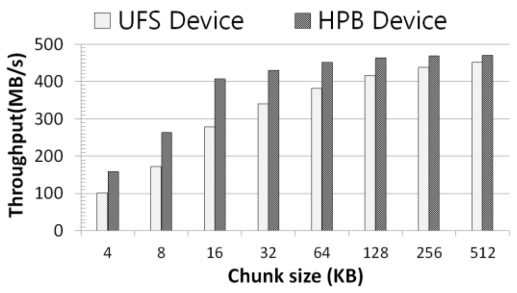

Figure 9: Performance for various chunk sizes.

Figure 7 shows there is no performance improvement in sequential read because the searching time for L2P map occupies only a small portion of the overall time. In addition, HPB does not affect write performance because there is no difference between HPB driver and normal driver under write workload.

Mixed workload: We tested read and write mixed workload to see HPB performance when L2P mapping changes due to host writes and host updates its L2P cache accordingly as described in 2.2.2. As shown in Figure 8, HPB achieves 8 ~ 43% performance improvements for various mix ratio. Even in highly write intensive load (read 10%, write 90%), HPB still gives 8% performance improvement.

Multi-page chunk read test: Figure 9 shows the result of random read workloads with various chunk sizes (4KB to 512KB). We tested with tiobench and set the test parameters to 1GB address range, 195MB I/O size, and 8 threads. As mentioned in 3.4, HPB driver delivers L2P entry if the requested chunk is physically contiguous. The performance gain of HPB becomes smaller as the chunk size increases because the elapsed time to load the L2P entry from flash memory becomes smaller compared to the time to read data.

5. Related Works

The concept of using the host-side memory to improve the performance of the device was proposed in NVMe Host Memory Buffer (NVMe HMB) [9] and UFS Unified Memory Extension (UFS UME) [10]. UFS UME requires additional hardware design for device to access host memory. U**boldface** FS device operates as a bus slave so it cannot access host-side memory by itself. Accordingly, additional interconnection interface is required for the UFS UME, which increases the hardware cost. In addition, UFS UME requires allocation of contiguous physical memory. In contrast, HPB offers significantly enhanced operational flexibility because it uses virtual memory allocated by device driver. The memory allocated for L2P cache can be returned to kernel and re-allocated for the HPB driver.

6. Conclusions and Future Works

In this paper, we propose HPB (Host Performance Booster), a method of caching Logical-to-Physical (L2P) mapping table in host-side memory to improve random read performance for storage devices that have not enough memory to keep the whole L2P mapping table in device's RAM. We implemented HPB on a Galaxy S7 with UFS (Universal Flash Storage) storage device. Experimental results show that HPB improves performance by 58 ~ 67% for random read workloads and 8 ~ 43% for read/write mixed workloads.

To our knowledge, our work is the first study on design and implementation of host managed L2P map caching. We hope our approach opens a variety of interesting research directions. Our future works include host-side L2P cache replacement algorithm and optimization of the L2P cache consistency protocol to reduce the L2P cache synchronization overheads. We also plan to implement HPB on NVMe SSDs and compare with the NVMe HMB.

7. Reference

[1] A. Ban. Flash file system, April 4 1995. US Patent 5,404,485.

[2] A. Ban. Flash file system optimized for page-mode flash technologies, August 10 1999. US Patent 5,937,425.

[3] Jesung Kim, Jong Min Kim, S.H. Noh, Sang Lyul Min, and Yookun Cho. A space efficient flash translation layer for CompactFlash systems. Consumer Electronics, IEEE Transactions on, 48(2):366-375, May 2002.

[4] Sang-Won Lee, Dong-Joo Park, Tae-Sun Chung, Dong-Ho Lee, Sangwon Park, and Ha-Joo Song. A log buffer-based flash translation layer using fully-

associative sector translation. ACM Trans. Embed. Comput. Syst., 6(3):18, 2007.

[5] J.-U. Kang, H. Jo, J.-S. Kim, and J. Lee. A Superb-lockBased Flash Translation Layer for NAND Flash Memory. In Proceedings of the 6th ACM & IEEE International conference on Embedded software (EMSOFT '08), Seoul, Korea, August 2006.

[6] S. Lee, D. Shin, Y.-J. Kim, and J. Kim. LAST: Locality-Aware Sector Translation for NAND Flash Memory-Based Storage Systems. In Proceedings of the International Workshop on Storage and I/O Virtualization, Performance, Energy, Evaluation and Dependability (SPEED2008), February 2008.

[7] A. Gupta, Y. Kim, and B. Urgaonkar. 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 XIV), pages 229–240, Washington, DC, March 2009.

[8] T.-S. Chung, D.-J. Park, S. Park, D.-H. Lee, S.-W. Lee, and H.-J. Song. System Software for Flash Memory: A Survey. In Proceedings of the 5th International conference on Embedded and Ubiquitous Computing (EUC '06), pages 394–404, August 2006.

[9] NVMe specifications 1.21a, http://www.nvmexpress.org/specifications/

[10] JEDEC JESD220-1A, Universal Flash Storage (UFS) Unified Memory Extension, Version 1.1. https://www.jedec.org/

[11] JEDEC JESD220C, Universal Flash Storage (UFS), Version 2.1. https://www.jedec.org/

[12] Samsung Galaxy S7. http://www.samsung.com/us/explore/galaxy-s7-features-and-specs/

[13] Samsung Universal Flash Storage. http://www.samsung.com/semiconductor/products/flash-storage/ufs/

[14] Threaded I/O tester. https://sourceforge.net/projects/tiobench/