공학석사학위논문

# SlimFTL: a Small and Fast Page-level FTL using Hash Functions

2020 년 8 월

서울대학교 대학원

컴퓨터공학부

신 재 민

# SlimFTL: a Small and Fast Page-level FTL using Hash Functions

지도 교수 김 지 홍

이 논문을 공학석사 학위논문으로 제출함
2020 년 8 월

서울대학교 대학원
컴퓨터공학부
신 재 민

신재민의 공학석사 학위논문을 인준함
2020 년 8 월

## Abstract

# SlimFTL: a Small and Fast Page-level FTL using Hash Functions

Jaemin Shin
Department of Computer Science & Engineering
College of Engineering
The Graduate School
Seoul National University

As the capacity of an SSD increases, the amount of DRAM for managing the SSD increases proportionally. Since the DRAM cost directly affects the overall SSD price, it is important to minimize the DRAM size without degrading the SSD performance. In this paper, we propose a novel hash-based FTL mapping technique, SlimFTL, that meets this goal. SlimFTL overcomes the GC inefficiency problem of an existing hash-based FTL in two directions. By employing an efficient indirection layer between the logical page and its hashed physical block, SlimFTL reduces the block copy overhead during GC. SlimFTL exploits the spatial sequentiality among successive writes so that sequential writes can be mapped to the same physical block, which significantly reduces the number of valid copies during GC. Experimental results show that SlimFTL can achieve the same performance level of a page−level mapping scheme with only 44% of the DRAM capacity.

**Keyword : hash−based mapping, page−level mapping, garbage collection cost, flash translation layer, solid−state disks**
**Student Number : 2018−25663**

# Contents

# List of Tables

# List of Figures

# Chapter 1. Introduction

## Section 1.1 Motivation

The capacity of a solid−state drive (SSD) keeps increasing by using more bits per cell like triple−level cell (TLC) [1, 2] and quad−level cell (QLC) [3, 4] and exploiting vertical dimension [5]. These advanced technologies make lower costs per bit and higher capacity per space for an SSD. Therefore, SSDs become the main storage device for datacenter. However, the large capacity SSD needs large internal DRAM to perform a flash translation layer (FTL). Especially, SSD maintains a mapping table in SSD's internal DRAM to support address translation from the host's logical page address (LPA) to SSD's physical page address (PPA). Since the mapping table size is related to the number of physical page addresses, its size is increased with the larger capacity of an SSD.

When the capacity of an SSD was small, the size of the mapping table was not an issue. So, FTLs generally have supported page−level mapping FTL(PFTL) which contains whole information in DRAM for flexible mapping management for random workloads. However, the size of the mapping table is becoming a problem as an SSD capacity is growing. By emerging technologies, the capacity of an SSD reaches 32 TB, which will require 16 GiB of DRAM in SSD to maintain the mapping table with an 8-KB mapping unit. To support higher capacity DRAM means higher costs and more space for DRAM. So, the use of large-capacity DRAM only for mapping information is a waste of space and cost within SSD. Therefore, it is necessary to reduce the amount of mapping information to be stored in the DRAM.

## Section 1.2 Contributions

Map−caching [6, 7, 8, 9] is the most commonly used method of reducing the size of the mapping table. This method stores entire mapping information in the flash chip and caches necessary information to DRAM. The map caching method has the advantage of being able to adjust the size of the mapping table by caching as much as necessary. In addition, exploiting the locality of requests access patterns can achieve a sufficiently high hit ratio on mapping information. However, there is a problem in that response time becomes longer when processing a request when a map−cache is missed. If the mapping information corresponding to the request needs to be loaded into the map−cache but the map−cache is full, some of the existing cached mappings must be selected as a victim and flushed with a flash chip to secure free space. Since this map−cache handling can take a long time, it can be a problem when processing a read request that requires a fast response.

Another way to reduce the mapping table is by using a hash function such as md5 [10] to efficiently manage mapping information. The advantage of this approach is that the entire mapping information is kept in DRAM, so there is no overhead of importing mapping information from flash chips such as the map−cache. However, FTL using a hash function is not widely used because it is difficult to resolve hash collision. Previously, HPFTL [11] was proposed, which addressed hash collision using the secondary table of the fully-associative method. However, HPFTL has a problem with excessive hash collision management overhead during garbage collection (GC), so it is not the only slow down the performance of an SSD but also reduces the lifetime of an SSD.

In this paper, we propose a method that uses the hash function but resolves the hash conflict more flexibly than the existing one. Our approach is based on the HPFTL approach, but we improve the existing technique in three major aspects. First, our approach handles hash collision more efficiently than HPFTL. When hash collision occurs in HPFTL, it stores corresponding mapping information into a log-based table which requires expensive reclaim process to make free entries. But our approach handles hash collision by redirecting hash result with a redirecting table which don't need any reclaim process.

Second, the GC overhead generated by a hash function is resolved using sequentiality. When a hash is used, sequential LPAs are stored in different blocks because of the randomness of a hash function. Sequential LPAs are likely to be invalid at the same time and these invalid pages are spread out, resulting in the uniform distribution of the invalid pages across the entire physical block. Uniform distribution of invalid pages makes it hard to choose a physical block with the high number of invalid pages during GC. Therefore, to improve GC efficiency, invalid pages need to be stored to the same physical block as much as possible, and we propose sequentiality-aware hasher to exploit sequentiality of LPAs and reduce GC overhead of proposed scheme.

Third, a shared virtual block is proposed to prevent hash collision even though its probability is very low when using our proposed scheme. If pointed virtual blocks using hash function are not able to be written, it can be solved by using the proposed scheme. It just writes to a virtual block which is adjacent to pointed virtual block like linear probing method generally used for a hash collision. Using the characteristics of the SSD, which checks whether page's LPA matches to request's LPA through out-of-band (OOB) when reading the page, the proposed scheme checks from original virtual block to adjacent virtual block until LPA in OOB of the pointed physical page is matched to request's LPA.

## Section 1.3 Thesis Structure

The rest of this paper is organized as follows. In Section 2, we review hash-based FTL and its fundamental problems which make it hard to use hash-based FTL. In Section 3, we present our design and implementation of SlimFTL in detail. Sections 4 and 5 describe our evaluation results and related work, respectively. We conclude in Section 6 with a summary.

# Chapter 2. Background

## Section 2.1 Overview of Hash-based FTL

Hash−based FTL is a method for reducing the large memory requirement of page mapping, and its purpose is the same as the map−cache method, but it does not cause any performance degradation due to map miss by managing all mapping table information in DRAM. An overview of a hash−based FTLs is shown in Fig. 1. Hash−based FTL reduces the mapping table size by managing encoded PPA using hash functions, unlike existing page mapping FTL which manages large size PPA for all LPA. Encoded PPA contains a hash function index and a offset in physical location. Through the hash function index included in the encoded mapping table, FTL calculates the hash result for a specific LPA using a hash function and decides a physical location like a physical block. After choosing a physical location, FTL chooses a PPA in physical location using the offset.

Since the physical location is determined using the hash function, it is essential to handling a hash collision. Hash−based FTL occurs the hash collision when physical locations pointed by hash functions are fulled and they are unable to write a new page. To guarantee an overall performance of hash-based FTL, hash collision should be handled efficiently. If collision handling is performed inefficiently, a hash collision occurs frequently and finally, it decreases overall performance of the FTL.

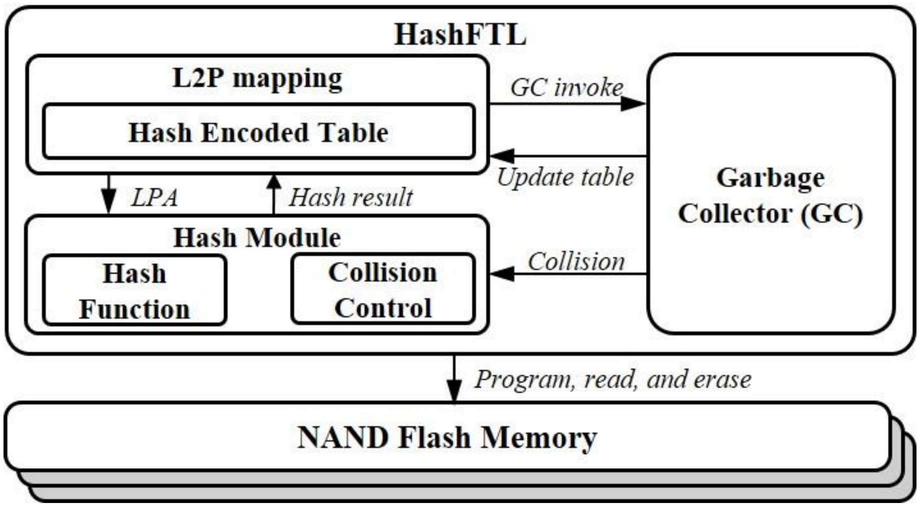

Figure 1. An overview of a hash-based FTL.

After each LPAs are hashed by function, LPAs are spread to random physical block chosen by the hash result. So, if sequential LPAs come to hash-based FTLs, it is stored in different physical blocks. Invalid pages are spread uniformly to all physical blocks because sequential LPAs are likely to be invalidated at the same time. This results in less invalid pages in victim block when performing GC and less free pages after GC finally more frequent GC to make enough free pages. Frequent

GC not only degrades FTL performance but also increases Write Amplification Factor (WAF) because of copying more valid pages. Therefore, hash-based FTLs should implement sequentiality-aware mapping to reduce GC overhead.

## Section 2.2 Existing hash−based FTL

HPFTL [11] is a hash−based FTL optimized mapping size using a hash function. HPFTL manages two mapping tables, a primary table, and a secondary table. The primary table manages the encoded PPA for each logical address and occupies most of the entire mapping size. Encoded PPA consists of a hash id (HID) and a partial page id (PPID). HID represents index information on which hash function was used to determine the physical block, and PPID is a value indicating where the page offset is in the physical block. When a read request for a specific logical address occurs, the target physical block is determined using the LPA as the input of the hash function corresponding to the HID of the encoded PPA, and a page indicated by the PPID for the physical block is selected.

The hash collision occurs in HPFTL when there is no space available in physical blocks selected through each hash function for a specific LPA, so it cannot be mapped through a primary table. In this case, the mapping is managed by referring to the secondary table by marking it as a specific signature in the primary table. The secondary table is fully−associative mapping which contains both LPA and PPA, therefore LPAs managed by the secondary table can be mapped to any physical page. However, because LPA and PPA pairs are managed together, the size of the secondary table entry is large, so it is necessary to manage only a limited number of entries in the secondary table. When the number of entries in the secondary table reaches a certain threshold, HPFTL reduces the number of secondary table entries by an entry reclaim process. The reclaim process allocates a free block and copies the data of LPAs that can be mapped to the free block through a hash function among LPAs managed with the secondary table to manage them as primary table entry.

HPFTL can successfully reduce DRAM usage compared to page mapping FTL due to the primary table using encoded PPA and the secondary table with the limited number of entries used under the hash collision cases. However, HPFTL has two serious problems compared to page mapping FTL. First, the overhead of hash collision management is too high. In particular, the secondary table is used to manage all valid pages copied during GC execution, which has a problem of rapidly increasing the number of GC executions by interacting with the reclaim process to manage the number of secondary entries below the threshold. This not only degrades SSD performance but also reduces the lifetime of an SSD. Second, HPFTL's mapping policy is sequentiality-oblivious mapping. Sequential requests are spread to different blocks in HPFTL, because of the randomness of the hash function. Therefore, the GC overhead of HPFTL is higher than that of PFTL.

## **Section 2.3** HPFTL Evaluation Result

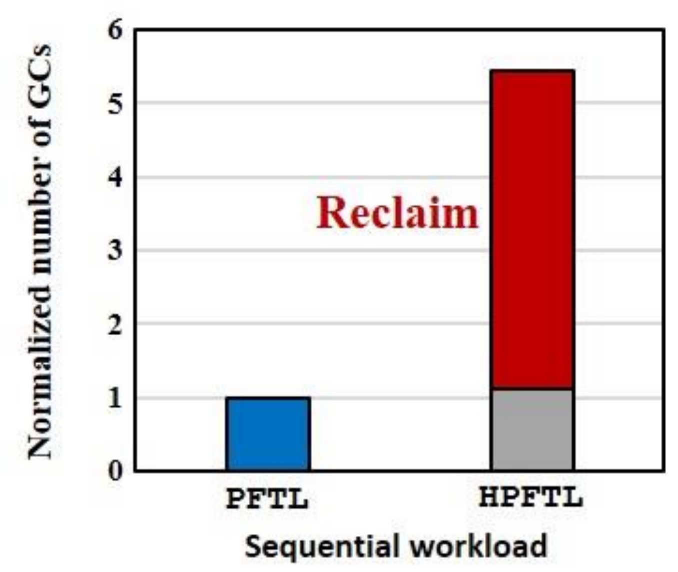

Figure 2. The number of GCs normalized to PFTL with sequential workload

We compared the GC overhead using two workloads with different update patterns to evaluate the characteristics of HPFTL that cannot exploit spatial locality and has excessive secondary table management overhead. Fig. 2 and 3 show the normalized GC overhead of HPFTL based on the number of GC executions of page mapping FTL (PFTL). It shows that HPFTL performs a much higher number of GC compared to PFTL in both workloads. In sequential workload, HPFTL invoked GC more than 5 times than PFTL as shown in Fig. 2. Furthermore, the reclaim overhead is substantial, as a large percentage of GC execution is caused by the reclaim process to secure the secondary entry. The GC overhead for non−sequential workloads is not as different from the PFTL as in the case of sequential workloads, but still increases by more than 50% over PFTL as shown in Fig. 3.

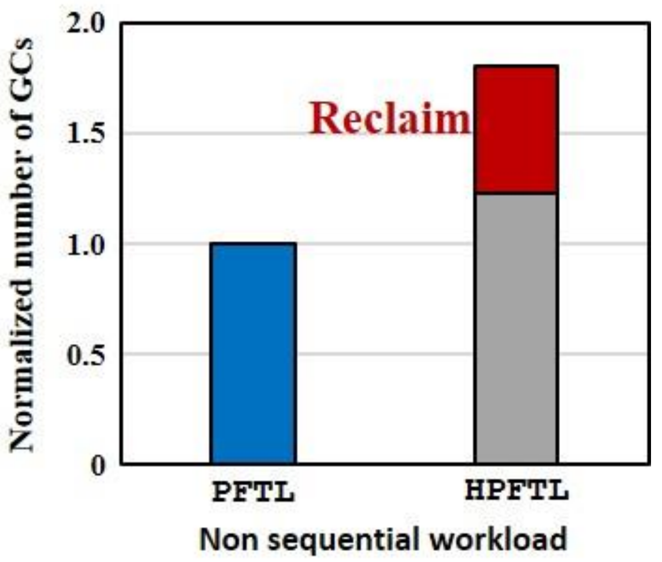

Figure 3. The number of GCs normalized to PFTL with non−sequential workload.

A fundamental reason why HPFTL has a high overhead is that it fixes LPA to certain physical blocks when using the primary table. This reason causes frequent use of the secondary table. Therefore, the primary table should make LPA not fix to certain physical blocks and it requires a new method with small overhead to solve the hash collision, unlike a secondary table method that increases GC overhead significantly. Also, it should make sequential-aware mapping because using hash breaks sequentiality and causes high GC overhead.

# Chapter 3. SlimFTL

## Section 3.1 Overview of SlimFTL

In this paper, we propose SlimFTL, which efficiently handles hash collision by using a block remapping scheme and uses sequentiality−aware hash scheme to minimize GC overhead in hash−based FTLs. An overall architecture of SlimFTL is shown in Fig. 4. A hash−based mapping table of SlimFTL consists of a primary table and a virtual block table. The primary table contains the encoded PPA for each logical page address, as same as in HPFTL. To reduce the probability of hash collision, SlimFTL additionally employs the virtual block table, which keeps the mapping between virtual block address (VBA) and physical block address (PBA). A sequentiality−aware hasher (sHasher) is responsible for generating VBA from LPA by using hash functions. In order to improve GC efficiency, sHasher exploits the spatial sequentiality among successive writes so that sequential writes can be mapped to the same physical block. Although the flexible mapping with indirection layer, a hash collision can rarely occur when all pages in the destination blocks are valid. A hash collision handler resolves this type of hash collision by sharing the virtual blocks which motivated by linear probing. In this section, we explain how SlimFTL's mapping process is operated. Then, we explain how the hash collision handler resolves the hash collision with a shared block scheme.

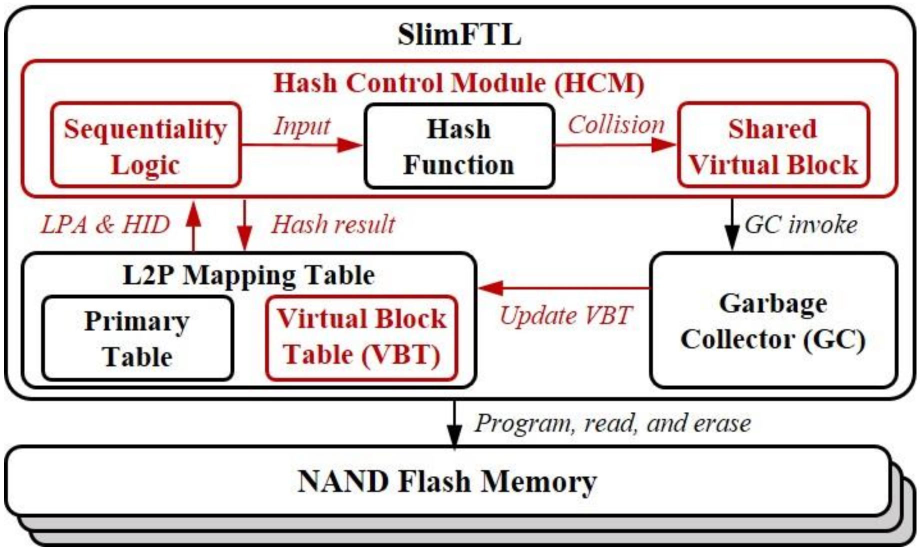

Figure 4. An overall architecture of SlimFTL.

## Section 3.2 Hash–based Mapping Table

The hash–based L2P mapping table of SlimFTL is consists of the primary table and the virtual block table. Each entry of the primary table keeps the hash function index (HID) and the physical page index (PPID). Therefore, the size of the primary table depends the number of used hash–functions and the number of pages in the flash block. For example, our default settings which use 64–hash functions and the number of page is 256, the primary table uses 14 bits per entry which uses 6 bits for HID and 8 bits for PPID. The number of entries in the primary table is the same as PFTL. The virtual block table uses a VBA as an index which is calculated from the hash function. The virtual block table uses 32 bits per entry and the number of entries is the same with the number of physical blocks which means it is a block–level mapping and negligible table size compared to the primary table.

The HID field of each entry is initialized to 0 which indicates that no writes have been come from each LPA yet. The physical block address (PBA) in the virtual block table is initialized to –1. If a write request comes to SlimFTL, it searches HID from 1 to 63 and stops if a VBA chosen by an HID has free pages. If an entry with given VBA is –1, SlimFTL allocates a PBA to the corresponding VBA and uses HID with this VBA.

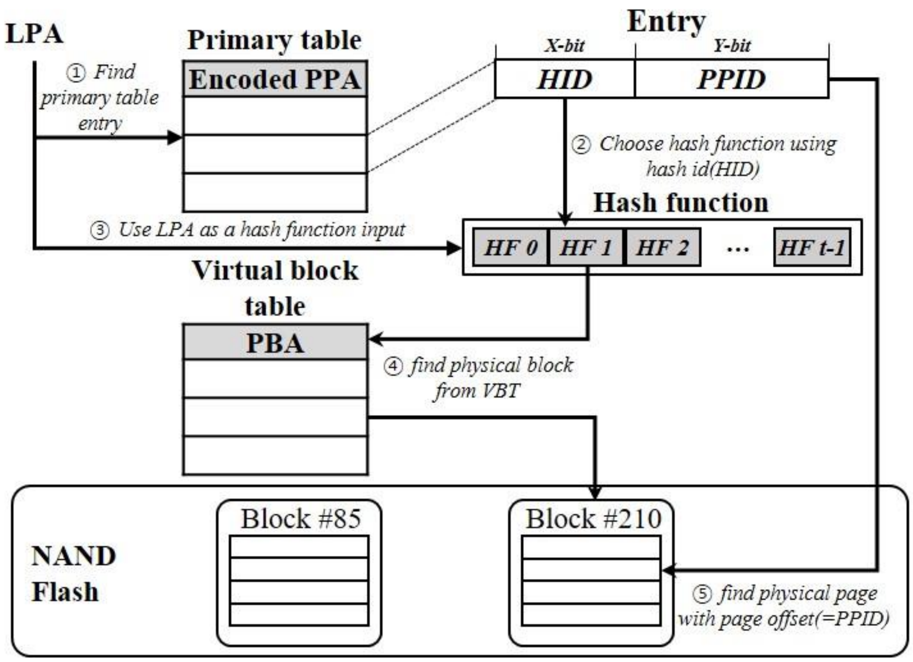
LPA
① Find primary table entry
Primary table
Encoded PPA
Entry
X-bit
Y-bit
HID
PPID
② Choose hash function using hash id(HID)
Hash function
③ Use LPA as a hash function input
HF 0
HF 1
HF 2
…
HF t-1
Virtual block table
PBA
④ find physical block from VBT
NAND Flash
Block #85
Block #210
⑤ find physical page with page offset(=PPID)

Figure 5. Virtual block scheme request flow.

The example of handling a request with the hash–based mapping table of SlimFTL is shown in Fig. 5.First, it reads an entry from the primary table using LPA as an index and the entry contains HID with 1 and PPID with 3. Using HID 1, it chooses hash function 1 and calculates hash results with LPA and hash function. Hash result means VBA and SlimFTL accesses the virtual block table with this VBA as an index. By using the virtual block table, SlimFTL translates the VBA to the PBA and chooses the physical block with corresponding PBA. From the physical block, a page is chosen by using PPID, and the page's PPA is the translated address of given LPA.

The virtual block scheme is used to redirect hash results. The hash result pointed to a physical block address, which is changed to a virtual block address. And we add the virtual block table indexed by virtual block address to convert virtual block address to physical block address. The virtual block table is mapping in block-level which means the number of entries in virtual block table is same with the number of physical blocks, it does not affect the overall size overhead by using only 0.4% of the size relative to PFTL.

The virtual block scheme can eliminate the hash collision during GC. Although GC changes the physical block address by copying valid pages to the reserved free block, updating the physical block address in the virtual block table to the physical block address of the free block table. Therefore, all valid pages in the block can point new physical block using HID in the primary table.

## **Section 3.3** Sequentiality-Aware Hasher

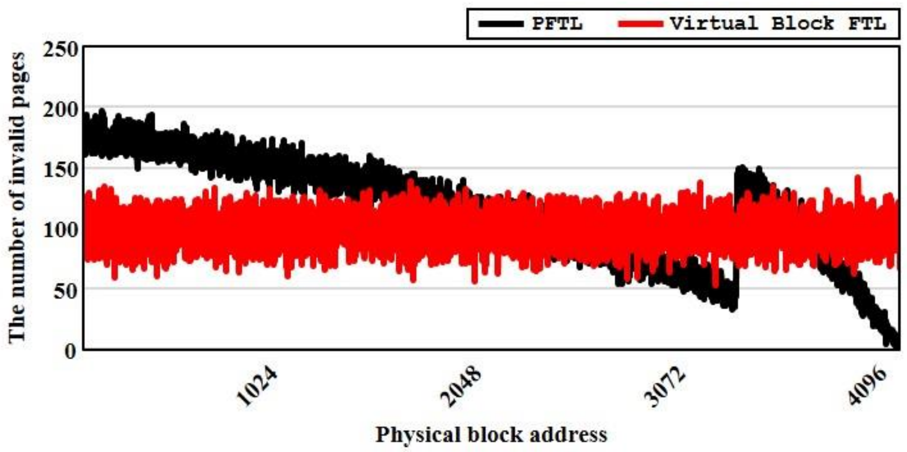

Figure 6. Invalid pages distribution when 1st GC occurs.

sHasher performs hash related operations in SlimFTL. First, SlimFTL changes input LPAs to more efficient form which can exploit sequentiality. Especially, in hash-based FTL, sequential writes are written to different physical blocks because of the randomness of the hash results. Even SlimFTL uses the virtual block table, the inefficiency in GC remains because successive writes spread multiple physical block. As shown in Fig. 6, we measured the number of invalid pages in each block in the hash-based FTL using virtual block table (Virtual Block FTL) compared to PFTL. Compared to PFTL, Virtual Block FTL shows that invalid pages are uniformly distributed. As a result, a victim block selected when performing GC has a small number of invalid pages. This in turn increases the number of valid pages that should be copied during GC and allows only small free pages to be obtained, resulting in more frequent GC calls and increasing overhead of hash-based FTL. This overhead is much larger than PFTL, and high overhead makes it hard to replace PFTL. Therefore, the overhead should be reduced and we exploit the sequentiality of requests.  In order to map successive writes to the same physical block, SlimFTL shifting least significant bits of LPA.

After shifting the LPA, 128-bits hash results calculated by the md5 [10] hash function is returned to the hash-based mapping table as a VBA. And SlimFTL shifts the results by the amount of the HID value. Through shifting the results, SlimFTL only needs to calculate md5 hash function for one time which reduces latency and it is possible because md5 outputs 128-bits hash results. From the shifted result, it is divided by the number of physical blocks and the divided result indicates VBA.

The proposed method to exploit the sequentiality of request is shifting LPA bit when it is input to the hash function. If the LPA bit is shifted, the sequential LPAs will use the same hash value. For example, if there are LPAs with 4, 5, 6, and 7, expressed in bits, it would be 100, 101, 110, 111. If FTL uses each as an input for the hash function, FTL will select a completely different block. However, if you shift 1 bit to the right, the bit becomes 010, 010, 011, and 011. LPAs 4 and 5 have the same result, and LPAs 6 and 7 have the same result. Therefore, each sequence can point to the same block. Through exploiting sequentiality, SlimFTL can collect invalid pages to few blocks. Thus, when performing a GC, block with many invalid pages can be selected, reduces the number of occurrences of GC and the management overhead of FTL.

## Section 3.4 Hash Collision Handler

In the proposed SlimFTL, hash collision rarely occurs because most of the hash collision in hash-based FTLs are occurred by copying valid pages during GC and it solved in SlimFTL by virtual block table. If corresponding VBAs are full, SlimFTL selects a VBA which has the maximum number of invalid pages and performs GC. However, if chosen VBAs are filled with valid pages, LPA cannot be written to those VBAs even if GC is performed.

To solve hash collision, a shared virtual block is proposed which solves the hash collision using linear probing which is generally used for solving hash collision. By using the proposed scheme, we just write it to a VBA which is next to the pointed VBA. Then, when reading, read the page from the selected VBA to the next VBA until the LPA in OOB of the page matches the request's LPA. This is possible because an SSD checks the LPA stored in the OOB area when it reads the page. Because it should read several times if it using the shared virtual block scheme, this reduces read performance. Basically, however, the probability of hash collision from this situation is very small, so performance is rarely reduced.

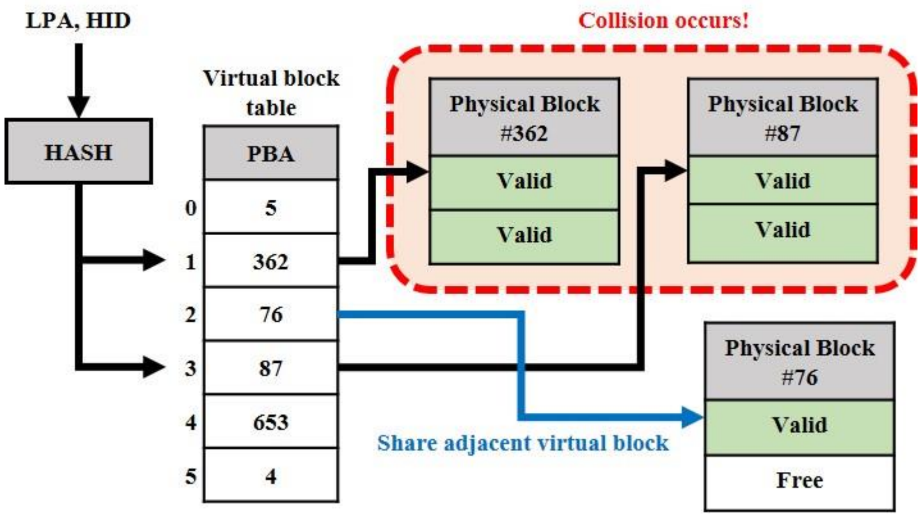

Figure 7. Shared virtual block

Fig. 7 shows when SlimFTL uses shared virtual block. Hash results are calculated by using LPA and HID and SlimFTL accesses virtual block table using these results as indexes. Using index 1 and 3, VBAs are translated to PBA 362 and 87. But at this moment, physical blocks are filled with valid pages therefore they cannot make free pages even if they are selected as a victim block and GC occurs. This is a hash collision situation and SlimFTL uses the shared virtual block to use the next virtual block. In this case, virtual block 2 is chosen as the shared virtual block of virtual block 1. Virtual block 2 points physical block 76 and it has free pages so SlimFTL writes to one of these free pages and updates the HID of the primary table to point original VBA which is the start point of shared virtual block address.

## Section 3.5 Garbage Collection

Garbage Collection in SlimFTL is slightly changed because of the virtual block table. GC with the virtual block table is invoked when virtual blocks that can be pointed by LPA are full. When GC occurs, a virtual block which can be chosen by GC invoking LPA is selected in the greedy policy. Next, it finds the PBA from the virtual block table. Then copy valid pages in the chosen block to a free block. Last, it updates the virtual block table information of the selected VBA to point the PBA of the free block.

During copying valid pages in victim block, it checks pages OOB and finds LPA of pages. Using these LPAs, SlimFTL updates the primary table belong to LPAs. For the HID field, the virtual block is the same because the changed physical block is updated in the virtual block table. For the PPID field, it is changed to a new location in the changed physical block. So, LPAs use the same HID to get a virtual block and use updated PPID to get PPA.

# Chapter 4 Experiments

## Section 4.1 Experimental Setup

In order to evaluate the effectiveness of the proposed SlimFTL scheme, we implemented SlimFTL on a custom flash storage system which is a simulation-based system. We used our flash storage system with a 16-GB storage capacity with a 10% overprovisioning capacity. As we used 256 8-KB pages per block, there are 9011 blocks in our flash storage system. To evaluate the effectiveness of the proposed scheme, we evaluated PFTL, SFTL, HPFTL, and SlimFTL using I/O trace generated from DBbench, TPC-C, YCSB-A, Varmail, and Fileserver. We used 6 bits HID and 8 bits PPID for the primary table. If we normalized the mapping size of each FTL based on the size of PFTL as 1, the mapping table size of HPFTL is 0.537, which is divided into 0.437 as the primary table and 0.1 as the secondary table. SlimFTL was evaluated as the condition having the same primary table size as HPFTL, and due to the virtual block table size of 0.004, the total size of the mapping was 0.441. SFTL was set to a level of 0.435 to have a mapping size similar to SlimFTL. By using these traces, we evaluated GC overhead and WAF value to verify the effect of the virtual block table and a sequentiality−aware hasher. To show the effectiveness of sequentiality−aware hasher, we evaluate the number of GCs for SlimFTL with changing sequentiality from 2 to 256 which means LPA is shifted by 1 bit to 8bit. If sequentiality is not mentioned, LPA is shifted 8 bit. All measurement results were normalized over a PFTL.

Table 1. SSD configuration for experiment

| SSD configuration | | | |
|---|---|---|---|
| SSD capacity | 16 GB | Pages per block | 256 |
| Overprovision area | 10% | Page size | 8 KB |

## Section 4.2 Evaluation Results

Fig. 8 shows the normalized IOPS of FTLs for each workload. The proposed SlimFTL shows the same performance to PFTL unlike HPFTL for all workloads because it uses efficient conflict handling mechanism virtual block table which needs only small overhead to manage and exploiting sequentiality of workloads which reduces the GC overhead. However, HPFTL shows only 38% on average of four workloads over PFTL. Because HPFTL's reclaim process causes lots of overheads and HPFTL uses sequentiality−oblivious mapping which makes more frequent GCs to generate free pages.

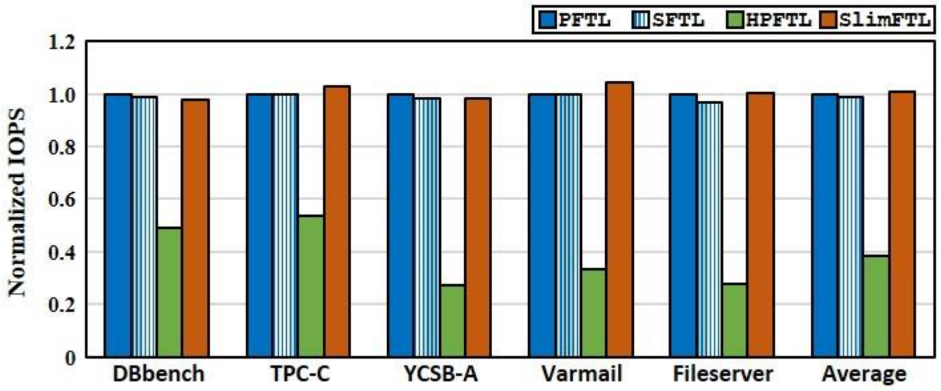

Figure 8. Normalized IOPS.

Fig. 9 shows the normalized number of GCs over the PFTL. For sequential write workloads DBbench and YCSB−A, HPFTL shows high

GC because it uses sequentiality−oblivious mapping which makes less free pages during GC. This results in more frequent GC than other FTLs. For other workloads, HPFTL needs additional GC due to reclaim process so it shows larger GC than other FTLs. But SlimFTL uses sequentiality−aware mapping with no reclaim process and it results in similar GC with PFTL. For non−sequential workloads, HPFTL's GC is still much higher than other FTLs because of the reclaim process. More GCs and additional page copy during the reclaim process cause higher WAF than other FTLs as shown in Fig. 10 which significantly decreases SSD's lifetime.

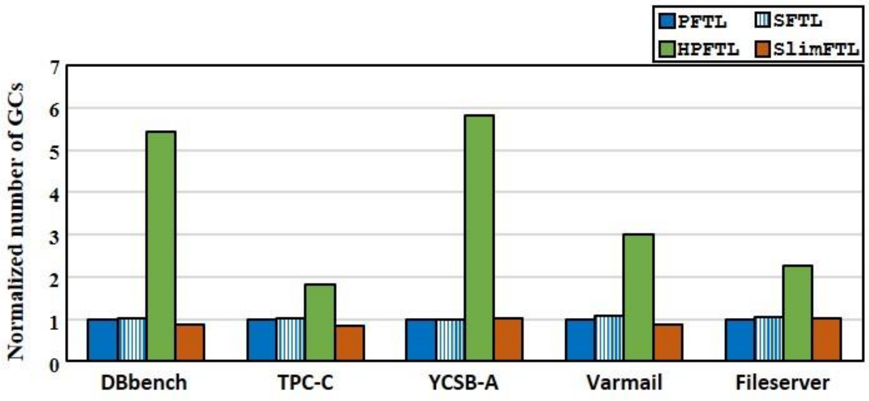

Figure 9. Normalized number of GCs for workloads

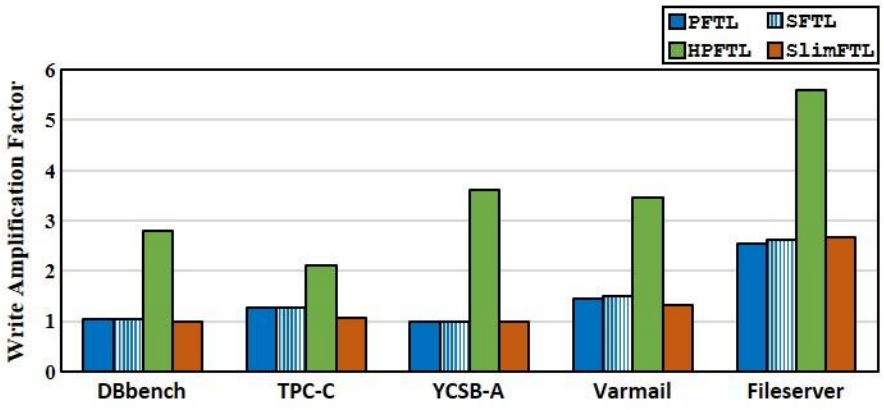

Figure 10. WAF for workloads

Fig. 11 shows normalized GC when sequentiality changes on DBbench workload. As sequentiality increases from 2 to 256, SlimFTL shows the lower number of GCs. Sequentiality with 2 pages shows 2.6 times for PFTL but sequentiality with 256 pages shows 0.87 times for PFTL which means SlimFTL with 256 sequentiality is even better than PFTL. Sequentiality with 256 pages can separate LPAs efficiently because it is the same as the number of pages per block and shows the same effect of hot−cold separation. So, SlimFTL can make physical blocks with 256 invalid pages and this results in copying no valid pages during GC and generating maximum free pages by GC. As a result, the number of GC is dramatically reduced by exploiting sequentiality.

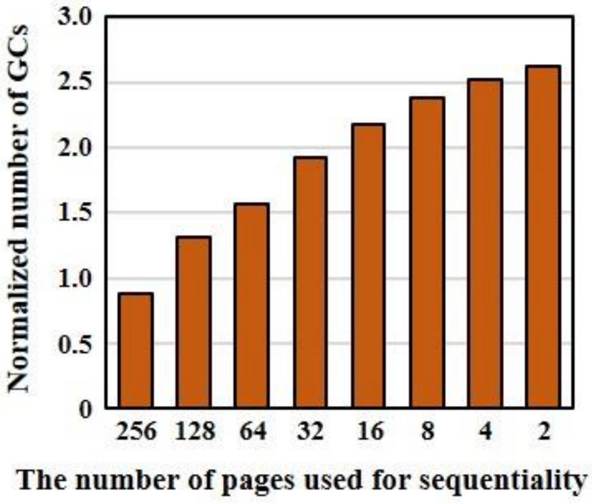

Figure 11. Normalized number of GCs when sequentiality changed

Because SFTL uses enough DRAM size, it shows a high hit ratio for most workloads. As a result, it shows a similar read latency to SlimFTL for these workloads. But if the workload has low locality, its hit ratio is lower and results in high read latency. Fig. 12 shows read tail latency of SFTL and SlimFTL for Varmail. SFTL shows higher tail latency than SlimFTL. In Varmail trace, the read hit ratio of SFTL is under 90% and SFTL suffers with high read tail latency.

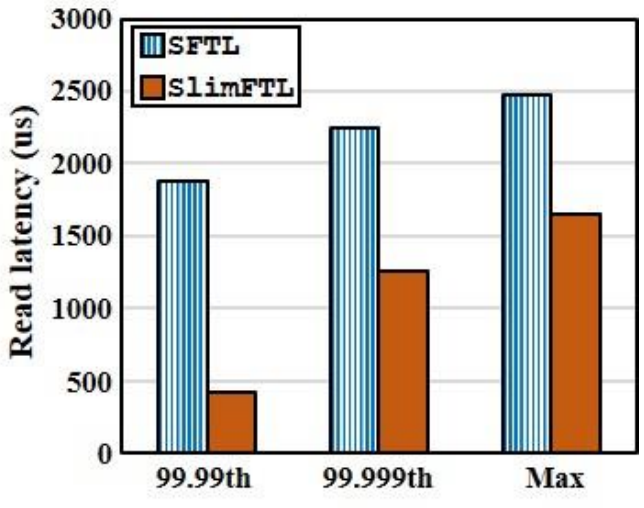

Figure 12. Read tail latency for SFTL and SlimFTL

# Chapter 5. Related Works

## Section 5.1 Related Works

There have been several studies to reduce the mapping table size of flash−based storage systems. However, most of these existing technologies are map caching−related techniques, so there is a problem of performance degradation when map−cache misses. DFTL [6] first proposed a map caching based method to reduce the mapping table size in DRAM. But if the workload has no locality, it results in lower performance. SFTL [7] used pages as a unit to manage mapping entries. It exploited spatial locality by enabling the prefetching of mapping entries. TPFTL [9] clustered the cached entries of each cached translation page in the form of a translation page node called the TP node. And It employed the two−level LRU lists to organize cached mapping entries. SHRD [8] maintained a part of mapping information in the host memory and used mapping information of host memory as a redirection table for random write performance. CAST [12] compacted the mapping table for sequential requests. It added a count column in entries and managed a range of the LPA mapping information. However, if the incoming request is not a sequential pattern, only the mapping table size increases due to the count column without performance improvement. HPFTL [11] proposed a first hash−based FTL with no performance degradation due to map−cache miss, but hash collision management overhead during GC is large and there is a limitation that workload locality cannot be utilized. Our technique is based on HPFTL, but it optimizes the hash collision issue during GC and reduces the GC overhead by utilizing host workload locality.

# Chapter 6. Conclusions

## Section 6.1 Summary

We have presented a novel hash-based FTL mapping technique, SlimFTL, to optimize DRAM memory size without degrading the SSD performance. By employing an efficient indirection layer between the logical page and its hashed physical block, SlimFTL reduced the block copy overhead during GC. Furthermore, SlimFTL exploited the spatial sequentiality among successive writes so that sequential writes can be mapped to the same physical block, which significantly increased the possibility of selecting a better victim block with a small valid page count when performing GC. We implemented SlimFTL on a custom flash storage system to verify the effectiveness of the proposed schemes. Our experimental results show that SlimFTL can achieve the same performance level as a page-level mapping scheme with only 44% of the DRAM capacity.

## Section 6.2 Future Work

The current version of SlimFTL can be further improved in several directions. For example, if we can utilize the released program sequence order of NAND, the size of the encoded mapping table can be further reduced. The PPID uses the number of bits that can point all location in a block because hash collision caused by program sequence shows high proportion when the PPID cannot point to all location. But if the program sequence order of NAND is released which can be done by 3D NAND, we can reduce the number of bits of PPID and result in more reducing in the size of mapping table.

# Bibliography

[1] H. Nitta, T. Kamigaichi, F. Arai, T. Futatsuyama, M. Endo, N. Nishihara, T. Murata, H. Takekida, T. Izumi, K. Uchida, T. Maruyama, I. Kawabata, Y. Suyama, A. Sato, K. Ueno, H. Takeshita, Y. Joko, S. Watanabe, Y. Liu, H. Meguro, A. Kajita, Y. Ozawa, Y. Takeuchi, Hara T., T. Watanabe1, S. sato, H. Tomiie, Y. Kanemaru, R. Shoji, C.H. Lai, M. Nakamichi, K. Owada, T. Ishigaki, G. Hemink, D. Dutta, Y. Dong, C. Chen, G. Liang, M. Higashitani, and J. Lutze. Three bits per cell floating gate NAND flash memory technology for 30nm and beyond. In 2009 IEEE International Reliability Physics Symposium (IRPS).

[2] Woopyo Jeong, Jae-woo Im, Doo-Hyun Kim, Sang-Wan Nam, Dong-Kyo Shim, Myung-Hoon Choi, Hyun-Jun Yoon, Dae-Han Kim, You-Se Kim, Hyun-Wook Park, Dong-Hun Kwak, Sang-Won Park, Seok-Min Yoon, Wook-Ghee Hahn, Jin-Ho Ryu, Sang-Won Shim, Kyung-Tae Kang, Jeong-Don Ihm, In-Mo Kim, Doo-Sub Lee, Ji-Ho Cho, Moo-Sung Kim, Jae-Hoon Jang, Sang-Won Hwang, Dae-Seok Byeon, Hyang-Ja Yang, Kitae Park, Kye-Hyun Kyung, and Jeong-Hyuk Choi. A 128 gb 3b/cell v-nand flash memory with 1 gb/s i/o rate. IEEE Journal of Solid-State Circuits 51, 1 (2015), 204–212.

[3] Cuong Trinh, Noboru Shibata, Takeshi Nakano, Mikio Ogawa, Jumpei Sato, Yoshikazu Takeyama, Katsuaki Isobe, Binh Le, Farookh Moogat, Nima Mokhlesi, Kenji Kozakai, Patrick Hong, Teruhiko Kamei, Kiyoaki Iwasa, J. Nakai, Takahiro Shimizu, Mitsuaki Honma, Shintaro Sakai, Toshimasa Kawaai, Satoru Hoshi, Jonghak Yuh, Cynthia Hsu, Taiyuan

Tseng, Jason Li, Jason Hu, Ming T. Liu, Shahzad Khalid, Junliang Chen, Mitsuyuki Watanabe, Hung-Szu Lin, Junhui Yang, K. McKay, Khanh Nguye, Tuan Pham, Y. Matsuda, K. Nakamura, Kazunori Kanebako, Susumu Yoshikawa, W. Igarashi, A. Inoue, T. Takahashi, Y. Komatsu, C. Suzuki, Kousuke Kanazawa, Masaaki Higashitani, Seungpil Lee, T. Murai, K. Nguyen, James Lan, Sharon Huynh, Mark Murin, Mark Shlick, Menahem Lasser, Raul Cernea, Mehradad Mofidi, K. Schuegarf, and Khandker Quader. A 5.6 MB/s 64Gb 4b/cell NAND flash memory in 43nm CMOS. In IEEE International Solid-State Circuits Conference (ISSC).

[4] Jung H Yoon, Ranjana Godse, Gary Tressler, and Hillery Hunter. 2017. 3D-NAND scaling and 3D-SCM—implications to enterprise storage. Flash Memory Summit 3, 4.2 (2017), 3–4.

[5] C. Kim, D. Kim, W. Jeong, H. Kim, I. Park, H. Park, J. Lee, J. Park, Y. Ahn, J. Lee, J. Lee, S. Kim, H. Yoon, J. Yu, N. Choi, Y. Kwon, N. Kim, H. Jang, J. Park, S. Song, Y. Park, J. Bang, S. Hong, B. Jeong, H. Kim, C. Lee, Y. Min, I. Lee, I. Kim, S. Kim, D. Yoon, K. Kim, Y. Choi, M. Kim, H. Kim, P. Kwak, J. Ihm, D. Byeon, J. Lee, K. Park, and K. Kyung. 2018. A 512Gb 3b/cell 64-stacked WL 3D-NAND flash memory. IEEE Journal of Solid-State Circuits 53, 1 (2018), 124–133.

[6] Aayush Gupta, Youngjae Kim, and Bhuvan Urgaonkar. 2009. DFTL: a flash translation layer employing demand-based selective caching of page-level address mappings. Acm Sigplan Notices 44, 3 (2009), 229–240

[7] Song Jiang, Lei Zhang, XinHao Yuan, Hao Hu, and Yu Chen. 2011. S-

FTL: An efficient address translation for flash memory by exploiting spatial locality. In IEEE 27th Symposium on Mass Storage Systems and Technologies (MSST).

[8] Hyukjoong Kim, Dongkun Shin, Yun Ho Jeong, and Kyung Ho Kim. 2017. {SHRD}: Improving Spatial Locality in Flash Storage Accesses by Sequentializing in Host and Randomizing in Device. In 15th USENIX Conference on File and Storage Technologies (FAST).

[9] 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 the Tenth European Conference on Computer Systems (EuroSys).

[10] Wikipedia. 2017. MD5 — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/wiki/MD5

[11] Fan Ni, Chunyi Liu, Yang Wang, Chengzhong Xu, Xiao Zhang, and Song Jiang. 2017. A hash-based space-efficient page-level FTL for large-capacity SSDs. In International Conference on Networking, Architecture, and Storage (NAS).

[12] Zhiyong Xu, Ruixuan Li, and Cheng-Zhong Xu. 2012. CAST: A page-level FTL with compact address mapping and parallel data blocks. In IEEE 31st International Performance Computing and Communications Conference (IPCCC).

# 초 록

SSD의 용량이 크게 증가하면서 SSD의 내부 데이터 관리를 위해 더 큰 용량의 DRAM이 필요하게 되었다. DRAM의 비용이 SSD의 전체적인 비용을 결정하는 중요한 요소이기 때문에 성능의 감소 없이 DRAM의 비용을 줄이는 것이 중요하다. 본 논문에서 우리는 이 목표를 달성하는 새로운 해시 기반의 기법인 SlimFTL을 제안한다. SlimFTL은 두가지 측면에서 기존 해시기반 기법의 GC 비효율성 문제를 극복한다. 논리주소의 해시 결과로 가리키는 물리 블록 사이에 리다이렉트 계층을 추가하여 가비지 컬렉션 동안의 복사 오버헤드를 감소시켰다. SlimFTL은 연속적인 쓰기의 연속성을 활용하여 같은 물리 블록에 쓰여질 수 있게 하여서 가비지 컬렉션 동안의 유효 페이지 복사의 수를 크게 줄였다. 실험 결과는 SlimFTL이 PFTL의 44%에 해당하는 DRAM 사용량을 보이면서 PFTL과 비슷한 수준의 오버헤드를 갖도록 줄였다.

**주요어 : 해시기반 맵핑, 페이지 레벨 맵핑, 가비지 컬렉션 비용, 플래시 변환 계층, SSD**
**학 번 : 2018-29102**