FTL Design for TRIM Command *

Joohyun Kim Haesung Kim Seongjin Lee Youjip Won

Department of Electronics and Computer Engineering
Hanyang University, Seoul, Korea
{createmain|matia|james|yjwon}@ece.hanyang.ac.kr

ABSTRACT

Introduction of SSD (Solid State Drive) to the market place has improved performance of PC dramatically because SSD has fast I/O processing capability, is shock resistant, and has no mechanical movements. TRIM command that notifies information of unused sector is suggested to ATA standard Technical Committee T13 to avoid overwrites and to handle invalid data effectively. We propose that in order to enhance the performance of the SSD up to enterprise standards, the Flash Translation Layer (FTL) has to support parallelism and TRIM command properly. We show that depending on the implementation of parallelism in FTL, write amplification factor can be an issue, especially on Hybrid Mapping FTL. In this paper, we introduce essential data structures to support TRIM command in FTL under multi-channel SSD architecture, and also address how and when the controller exploits this information. While exploiting the real-time evaluation and measurement system, Full System Emulator, we study the effect of the TRIM command implemented in two modified FTL schemes: Hybrid Mapping and Page Mapping FTL. We also provide a model to measure the effectiveness of the FTL. Experiment of installing OS shows that TRIM command generally increases the performance by 2% in Hybrid Mapping FTL, and 13% in Page Mapping FTL as compared to FTL without support for TRIM command. IOZone benchmark result shows that exploiting TRIM in Page Mapping FTL provides 10.69% increase in throughput for random write.

Categories and Subject Descriptors

D.4.2 [**Operating Systems**]: Storage Management; D.4.8 [**Operating Systems**]: Performance

General Terms

Design, Experimentation, Performance

Keywords

TRIM Interface, Performance Evaluation, SSD Emulator, FTL (Flash Translation Layer)

*This work was supported by IT R&D program(Large Scale hyper-MLC SSD Technology Development, No. 10035202) of MKE/KEIT.

1. INTRODUCTION

NAND Flash Memory is a nonvolatile storage semiconductor that constitutes a SSD as a basic element. Its advantages include fast I/O throughput, low power consumption, and shock resistance. However, NAND Flash Memory has its own demerits. Operation in SSD has different units. For example, erase operation affects a block, which is a group of pages, and a read/write operation affects a page. Another fact that a file system does not take account of is that cells in a NAND Flash Memory have limited number of erase counts. These characteristics hinder optimal performance of conventional file system. Therefore, hiding the inherent characteristics of NAND Flash Memory and emulating HDD interfaces in SSD is essential to exploit existing file system. To provide the same interface as the HDD and hide all the complication from OS, SSD uses a layer called Flash Translation Layer (FTL). The purpose of FTL is to map a sector onto a Page of NAND Flash Memory. Even sophisticated FTLs cannot completely provide solutions to the overwrite issue, asymmetric I/O, and the restricted number of write counts.

In most cases, overwrite operation is a bottleneck to performance of SSD because it issues two time consuming operations repetitively, namely erase and write operations. Thus, many researches focus on minimizing or optimizing overwrites in SSD based on NAND Flash Memory. Agrawal et al. [1] suggests that minimizing effect of overwrite operation can significantly improve the performance of SSD. A simple approach in avoiding overwrite of records is to direct the corresponding sector to a free area and modify corresponding mapping information. Remapping of the data not only reduces number of overwrite and erase operations but also distributes used sectors in NAND Flash Memory and wear levels; however, it causes an increase in the number of invalid sectors in NAND Flash Memory.

In this work, we aim to devise a novel data structure and algorithm to effectively and efficiently handle trim command [14]. The algorithm and the data structure proposed in this work are to be embedded at the SSD controller. TRIM enabled file system and Operating System inform SSD controller about "useless" blocks. Controller can exploit this information in erase and merge operation and avoids unnecessary block and page copy operation of "useless" blocks. SSD controller is responsible for storing incoming trim information efficiently and for taking proper action at the right time exploiting the information provided by trim command. This work aims at addressing two technical issues in depth. The first issue is the design of data structure for trim com-

mand. File system can send trim information in bursty manner. SSD controller should be able to store and search block deallocation information instantly. The second issue is how and when the controller exploits this information.

1.1 Motivation

Modern state of art SSD controller has much to gain from exploiting block deallocation information issued by TRIM command. Block deallocation information can be exploited in many operations, e.g., file deletion, application installation, OS boot, etc., where file system collects information on unused blocks and to-be deleted blocks. When passed to SSD controllers, SSD controller can exploit this information in efficiently managing the NAND Flash pages and NAND Flash blocks. To fully take advantage of block deallocation information, two constraints need to be addressed. First, application file system or Operating System should be designed to exploit TRIM command. Secondly, underlying SSD needs to be designed to incorporate block deallocation information in its FTL algorithm. Unfortunately, however, few applications actually exploit TRIM command. EXT4 claims that it adopts TRIM command, but according to our experiment, we found that EXT4 does not generate TRIM command at all. Windows 7 is one of the few softwares which actually uses TRIM command and passes block deallocation information to SSD. However, we carefully believe that this feature is not mature enough either to fully exploit the nature of SSD or to properly incorporate the limitation of currently used SSD.

Let us provide an example. According to our experiment, Windows 7 Operating System uses TRIM command and sends unused sector list to SSD in boot phase. For fresh file system partition of tens of GByte capacity, the number of sectors, whether they are consecutive or not, can easily go over tens of millions. Therefore, to exploit full potential of block deallocation information, file system should have capability for search of used sectors and SSD should be capable of harbor block deallocation command in search-efficient and insert-efficient data structure.

Exploiting parallelism is important in SSD because it brings accreted effect by amalgamating the performance of low bandwidth Flash Memories. Although parallelism of SSD is not an issue directly related to TRIM command, we implement parallelism in this work because all modern SSD adopts multi-channel and multi-way architecture to increase read and write bandwidth. Parallelism can be realized in a FTL by designing striping, interleaving, and pipelining functions [6]. Striping divides a request into sub-requests and delegates each sub-request to separate channels; interleaving technique makes channel managers responsible for assigning requests to each channel to earn parallelism; pipelining technique provides parallelism by sharing a single channel by processing command-processing phase in second request right after when first request is through with data transfer phase. In order to maximize the parallelism of SSD, all aspects of the system has to be considered such as characteristics of plane, Page size in I/O, Block size in Erase operation, and configuration of Bus; maintaining the parallelism is a key to acquiring high performance SSD.

There are three major contributions in this paper.

1. **Issues in implementing TRIM Interface** Although TRIM command is introduced to the research community, there is little studies on the effect of TRIM to storage system and to SSD. This paper presents insight in exploiting the interface. We integrate the interface with two of renowned FTL schemes, FAST [9] and Page Mapping FTL [2]. In the course of maximizing the performance of modified FTLs, we also propose data structures to effectively utilize the parallelism in the SSD architecture.

2. **Effect Factor of SSD** In this paper, we present a simple, yet intuitive measure to assess the performance of the FTL. It takes account of number of blocks used in programming requested data, used in programming requested sectors, and average erases that a program operation incurs.

3. **Full System Emulator** Unlike test driven experiments that does not take account of running system, our full system emulator can acquire real time information of Operating System, file system, and most importantly the Storage System. In this paper, we present the detailed view of the emulator and how two FTL architectures are integrated in the emulator.

Table 1: Specification of K9LAG08U0M. Sequential Read: 28.8MB/s (Transfer time: 7680 ns + Read time: 60 μs) Sequential Write: 2.4MB/s (Transfer time: 7680 ns + Program time: 800 μs) [13]

Features	Spec.	Features	Spec.
Capacity	16,384 Mbit	Read Page	60 μs
Program Page	800 μs	Erase Block	1.5 ms
Flash size	8192 blocks	Block Size	128 pages
Page Size	2048 Byte		

2. SYNOPSIS

2.1 NAND Flash Memory

This subsection introduces NAND Flash Memory model used in the paper. There are two types of a NAND Flash Memory, Single-Level Cell (SLC) and Multi-Level Cell (MLC). SLC type NAND Flash Memory can store one bit in a cell, and it has better read and write speed and endurance level compare to MLC type NAND Flash Memory; however, SLC is expensive and has less storage capacity than that of MLC. Since MLC type NAND Flash Memory can store two bits in a cell, MLC can store more data in same dimension. Experiments in this paper use MLC type NAND Flash Memory. Features and performance of NAND Flash Memory is based on K9LAG08U0M Model and key specifications such as read, program, erase operations, and other features are shown in Table 1.

2.2 SSD Architecture

In this subsection, we discuss the SSD architecture implemented in full system emulator. SSD has three main parts, host-side processing module, controller module, and set of NAND Flash Memory. Host-side processing module communicates ATA commands with controller in the storage system. Controller module, known as FTL in SSD, consists of functions to process read, program, and other requests to a NAND Flash Memory. All data are stored in the third component of SSD, the NAND Flash Memory. Location of DRAM controller is dependent on the architecture of the storage system; some include it inside the controller. We

develop full system emulator where we can implement multi-way/multi-channel SSD with various FTL algorithms, write buffer management schemes, error correction module, etc.

SSD used in this paper is comprised of four K9LAG08U0M based Flash memory with capacity of 8 GB, and its parallelism is maximized by placing independent bus and NAND controllers to each NAND Flash Memory. Host-side module translates ATA standard command requested from file system into command that a FTL can process, such as Read, Program, and TRIM. Programming data in a NAND Flash Memory comprised of three steps, command processing phase, data transferring phase, and data programming phase [11]. In this study, we install commodity OS (Windows 7) on our simulator and examine the detailed behavior of SSD with and without TRIM command. At the time of this study, we were not aware of the other OS that uses TRIM command. In initial phases, OS determines whether the underlying storage is SSD or HDD. IF it is determined to be SSD, then it checks whether the storage device is TRIM enabled SSD or not. File system used in Windows 7 exploits TRIM feature as soon as the OS finds out that attached storage system can process TRIM command. In our simulated SSD, each NAND Flash Memory in the SSD is connected to independent bus, and the SSD provides programming speed of 18.8 MB/s and read speed of 207.2 MB/s. NAND Flash Memory used in full system emulator has four planes. Size of DRAM is not limited in this work, and to be fair in measuring the effect of TRIM in FTL, additional features like caching is not implemented. Since TRIM command affects write and erase operations mostly, we do not consider its effect in read intensive workloads.

3. TRIM AND HYBRID MAPPING

In Hybrid Mapping scheme [7, 5, 9, 8], there are two types of blocks: Data and Log. For each type of blocks, that a Hybrid Mapping scheme uses two different Mapping schemes. In managing Data block, Hybrid Mapping scheme generally exploits Block Mapping, and Log Block is managed by Page Mapping scheme. This Hybrid scheme appreciates the small mapping table size compared to Page Mapping scheme and better I/O performance compared to Block Mapping scheme, but suffers from recurring Merge operations to move data on Log Blocks to Data Blocks. Copying contents of Log Blocks to Data Blocks are not always straightforward in SSD because it does not support in-place update; Merge operation checks both Data Block and Log Block, and copies only validate pages in each block to a clean Data Block, then erases old blocks.

Although many have proposed Hybrid-Mapping schemes, I/O propagation issue in Merge operation requires much attention from the research groups to minimize the effect. FAST scheme proposed by Lee et al [9]. FAST minimizes Merge operation frequent in Log Blocks of BAST [7] assigning exclusive properties to Log Blocks. One set of Log Block is dedicated to sequential requests and the other set is dedicated to random requests. By having dedicated sequential Log Blocks, the performance of sequential requests increases because cost of merge operation reduces. By having two different Log Blocks with dedicated purposes, implementing FAST becomes a better choice than BAST in terms of speed and endurance. However, the fact that FAST does not consider parallelism in processing write operation to Log Blocks it requires modifications to accelerate the I/O performance.

3.1 Hybrid Mapping and Multi-channel Architecture

FAST FTL in full system emulator could not handle the real time OS installation workload because it is not designed to operate on multi-channel/multi-way environment. FAST spends much of its time in merging and switching its log blocks, instead of processing the installation. One way to overcome the overhead of real time workload is to exploit the parallelism in the SSD architecture in FTL. How to write a data into a data block and log block has to be reconifigured to support striping, interleaving, and pipelining in FAST. Data is striped based on the plane and channels in NAND Flash Memory when a write request is issued. Since each NAND Flash Memory is connected to independent channels, we distribute consecutive write requests to each channel without explicitly exploiting the interleaving technique. There are two approaches in striping sectors. First approach is to strip it in units of page. The controller strips the request into consecutive sub-requests with length of sector adequate for single page, which is four since we use four channels in our implementation, and processing them upon receiving sub-requests in different NAND Flash Memory. The other approach can divide the same request into larger sub-requests, in units of block where each NAND Flash Memory receives 256 sectors. Programming in units of pages and blocks distributes the writing requests to many cells in the memory. The dispersion causes frequent occurrence of Merge operation, which calls for close attention.

There are two types of log block in FAST FTL: sequential log block and random log block. Adopting parallelism in random log block is not as difficult as sequential log block because random log block is only dependent on merge operation. On the other hand, sequential log block is dependent not only on merge operation but also on switch operation. A merge operation can cause increase in write amplification factor [4]. The write amplification factor depends on how parallelism in data block is implemented. Since a request is striped and distributed to separate sequential log blocks, it decreases the probability of a sequential log block being switched. We consider two data structures in implementing parallelism in the FTL, page-level and block-level parallelism.

In page-level parallelization shown in Fig. 1, controller partitions a write request into pages and evenly distributes them each channel. Using Page-level parallelism in a NAND Flash Memory greatly enhances speed of reading and programming because it divides a request and programs them into number of blocks in the Flash memory, and reads pages from number of blocks in the NAND Flash Memory all together. However, this method adversely induces associativity problem in performing Merge operation. For example, if a file system sends a write request of 1024 sectors, page-level parallelism in FTL distributes the request to all available random log blocks. When FTL decides to merge the 1024 sectors written in all random log blocks, FTL has to refer to corresponding random log blocks, and process the merge operation. As number of associated log blocks increase, I/O propagation also increases, which leads to longer processing time for Merge operation and at the same time degrading the performance.

In block level parallelization shown in Fig. 2, write request is partitioned into a unit of Flash Memory block and each

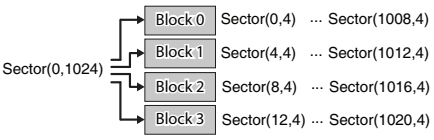


Figure 1: Page-level Parallelization

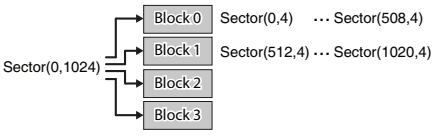


Figure 2: Block-level Parallelization

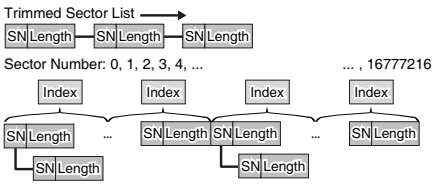


Figure 3: Data Structure for the Trimmed Sector list

of the blocks is allocated to different channel. If the size of write request is smaller than a block, the data is written to single channel and thus write operation cannot exploit the parallelism. In this work, we use block-level parallelism in address interleaving. We found that page-level parallelism makes the associativity of log block prohibitively large, especially under multi-channel architecture and therefore that hybrid mapping cannot be used with page level parallelism. Block-level parallelism delivers reasonable performance with much less overhead in log block merge compared to the overhead of log-block merge operation in hybrid mapping with page-level parallelism.

3.2 Adopting TRIM: Hybrid Mapping

TRIM command carries set of sector numbers and respective lengths that notifies the number of requested sectors. Once file system sends TRIM command, it does not send further I/O requests until storage system returns response. Upon receiving TRIM command, FTL has two choices. First is to process them as soon as receiving the request and send response to the file system. Second is to send response to the file system, but processes the request at a convenient time. In some cases, processing the TRIM command in real-time can be ineffective. For example, according to our experiment, Windows 7 issues more than millions of unused sector lists to storage system via TRIM command. It is not practically possible to perform erase operation for millions of sectors in real-time. Erase operation can hinder performance because discrepancies in operation speed of erase and read/write holds the performance behind, and I/O requests are not further processed until all TRIM command is properly handled.

In this work, we propose to maintain block deallocation information passed by TRIM in memory and to postpone its processing until the actual merge operations occur. The prime objective of this approach is to minimize the latency and the interference with the ongoing operations. For efficient search, insert, and delete operation, data structure for maintaining block deallocation command plays a key role. We propose hash based linked list (HBL), Fig. 3. In HBL, all sectors are partitioned into N sets, each of which contains consecutive sectors. Block deallocation information takes the form of [*Start No*, *Length*]. Incoming block deallocation information is inserted into appropriate group. If incoming block deallocation information lies across the group boundary, we partition it so that the resulting information lies within a group. The sectors within a group are sorted with sector number and are organized using linked list. Our implementation of TRIM sector lists in Hybrid Mapping FTL allows to exploit the list in the case of Merge operation, and also be utilized in enhancing the performance of Wear-leveling, and Garbage Collection (GC).

4. TRIM AND PAGE MAPPING

Page Mapping FTL maps the entire sector into pages of any Blocks. Exploiting parallelism through Bus and Flash Memories in Page Mapping FTL is easier because Page Mapping FTL does not require frequent Merge operation or overwrite operations, etc.

When Page Mapping FTL receives data for programming, it partitions the request to smaller sub-requests with four sectors that can be issued to a page of a block in plane. Striped sectors are programmed exploiting pipelining. Note that an even numbered block comprises even plane, and odds comprises odd plane. This configuration can produce six times faster OS installation time than Hybrid Mapping Scheme. Invalidated pages caused by overwrites are continuously monitored in units of blocks, and when all pages in a block is invalidated, corresponding blocks are erased. Behavior of GC is similar in terms of moving the data from one block to another and erasing the blocks, difference of GC from merging is that it begins only when remaining storage area is insufficient.

4.1 Adopting TRIM: Page Mapping

Since Page Mapping FTL does not entail Merge operations instead activates GC to procure sufficient free space in the storage, Page Mapping FTL does not have to manage the TRIM sector List used in Hybrid Mapping FTL. When the FTL receives TRIM command, it simply erases sectors from the storage.

We designed two types of Page mapping table to manage the unnecessary sectors in Page Mapping FTL as shown in Fig. 4. Forward Mapping table allows to search a page using the location of a sector, conversely Backward Mapping table locates sectors in a Page. Upon receiving a data to be programmed, both of the Mapping table is modified to reflect the changes. Backward Mapping table provides information on remaining number of valid sectors in each block. When all sectors in a Block in invalidated, the corresponding block is erased from Backward Mapping table. Sectors requested by TRIM command are only erased in the Forward Mapping table, which causes discrepancies in contents of two Mapping table. The difference is reflected on Backward Mapping table regularly. Page Mapping FTL activates GC process when remaining free space in SSD under 30 percent; GC process chooses blocks with least number of valid sectors as a victim. As the remaining storage area decreases, the frequency of GC process increases.

5. EXPERIMENTS

This section provides experiments and evaluation of performance and effect of TRIM command. Note that we disabled buffer cache management in the SSD to visualize the effect of TRIM command. Our implementation of SSD architecture supports four parallel NAND flash channels.

5.1 Full System Emulator with QEMU

This subsection describes full system emulator, which measures the performance of the OS and Storage system. The emulator allows processing the I/O requests from OS and Application in real time. Unlike other trace driven evaluation approaches, it provides holistic performance measurements to changes in storage system. Our system emulator is based on QEMU/KVM [3] as shown in Fig. 5.

Full system emulator adopts two aforementioned FTL schemes. An independently operating real time monitoring tool verifies the I/O, Merge, GC, etc in SSD, which have significant benefits over trace driven evaluation. Full system emulator generates a log file after each iteration of experiment to make record of wear-level of each NAND Flash Memory, and mapping table.

Current implementation of Hybrid FTL uses single log block per chip for sequential requests and four log blocks per chip for random requests. There are four chips configured in the SSD architecture. Block Status Monitor module gathers status of each block. TRIM manager manages all TRIM commands, and exploits the information when FTL performs Merge operation. Every request that TRIM manager receives are reflected in mapping table right away. Block Manager manages the status of each Block

5.2 Modeling FTL Effect Factor

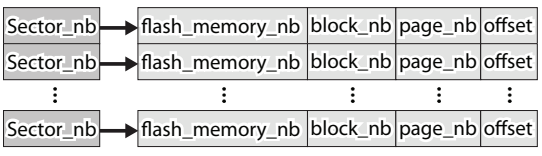
This subsection describes the performance result of Hybrid Mapping FTL and Page Mapping FTL on two experiments. First is installing Windows7 that implements TRIM command in file system and second is IOZone benchmark, which is standard in measuring the performance of a storage [10]. Then, we describe a model for assessing the performance of SSD.

Eq. 1 measures the effect of TRIM command, where number of valid blocks, N_{vb}, denotes number of blocks with valid data after all program requests in the experiment is processed. Number of programmed blocks, N_{pb}, denotes number of blocks programmed while processing the program requests in a experiment. Number of erased blocks, N_{eb}, denotes average erase counts of a block in a experiment.

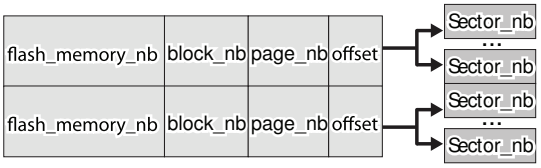
$$E_{ftl} = \frac{N_{vb}}{N_{pb} \times N_{eb}} \quad (1)$$

5.3 OS Installation

Table 2 shows performance result of Hybrid Mapping FTL in installing Windows7, with and without triggering TRIM. The command did not cause difference in time spent in installing the OS; duration of the installation lasted 41 minutes for both. Examining the result triggering TRIM right after installing the OS, E_{ftl} measures 53.0 and 51.9 for TRIM and without TRIM, respectively, where enabling TRIM gives slightly better effectiveness. Wear leveling of each block in Hybrid Mapping FTL with TRIM improves about 2% compared to Hybrid Mapping FTL without the support for TRIM command. Wear leveling of Page Mapping FTL with TRIM enhances about 11.6% compared to Page Mapping FTL without support for TRIM command. Wear-leveling measurement of Page Mapping FTL compared to Hybrid Mapping FTL produces about 60 times better performance. The result indicates that not only in I/O performance but also in wear leveling in Page Mapping is by far better than the hybrid method.



(a) Forward Mapping Data



(b) Backward Mapping Data

Figure 4: Page Mapping Data Types

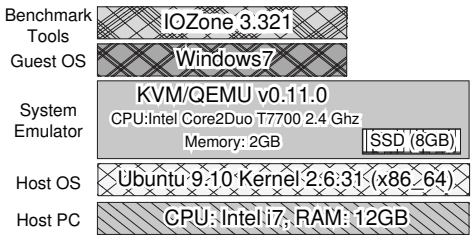


Figure 5: Full System Hierarchy

Table 2: Effect of TRIM: Hybrid vs. Page Mapping

	Hybrid Mapping		Page Mapping	
	TRIM	w/o TRIM	TRIM	w/o TRIM
N_{pb}^{OS}	53119	53655	23512	23480
N_{pb}^{IOZone}	32479	32221	2920	2112
N_{eb}^{OS}	1085	1102	20	23
N_{eb}^{IOZone}	1223	1247	103	92

Table 3 compares performance of two FTL scheme in installing Windows7. Page Mapping FTL outperforms Hybrid Mapping FTL both in number of programmed blocks, N_{pb}, and time spent in installing the OS. Hybrid Mapping FTL uses 2.3 times more blocks and 5.7 times longer period in installation. There are two main reasons behind the outstanding performance of Page Mapping FTL with support for TRIM command. First, there is no merge operation, which is recurrent in Hybrid Mapping FTL. Second, there is no GC process, which is activated when the free space in the storage is less than 30 percent. The result suggests that enterprise level SSD exploiting Page Mapping scheme can produce incomparably better performance.

5.4 IOZone Benchmark

This subsection describes the IOZone Benchmark result on Table 2. The effect of TRIM measured by E_{ftl} of Hybrid Mapping FTL with IOZone benchmark reads 79.3 and 78.5 for TRIM and without TRIM, respectively. Result shows that there are more random writes and overwrites in IOZone than in installing the OS. The result also indicates that TRIM enhances the efficiency of FTL and wear leveling in IOZone benchmark.

Table 4 shows effect of TRIM command in Page Mapping FTL. Enabling TRIM enhances 12.4% of write speed compared to FTL without TRIM capability. It also shows higher throughput on other measures; it enhances 13.5% of re-write and 10.7% of random write test. TRIM command in Page Mapping FTL helps to enhance the performance in short

Table 3: Hybrid vs. Page Mapping

Mapping	N_{pb}	Duration
Hybrid	53119	41 Min.
Page	23512	7 Min.

Table 4: IOZone: Page Mapping FTL (MByte/Sec)

	w/ TRIM	w/o TRIM	Improvement
Write	137.6	120.5	12.4%
Re-write	222.6	192.7	13.5%
Random Write	222.8	199.0	10.7%

term; however, as the storage suffers from deficiency of storage area, performance is severely affected by GC. The GC scheme runs when available storage area goes under certain threshold level. In our implementation of FTL, GC scheme is executed when available storage area is less than 30%. When GC is running under TRIM enabled Page Mapping FTL, the result of IOZone benchmark compared to TRIM disabled FTL gives 1.3% lower write throughput, 4.7% and 2.3% lower throughput for re-write and random write, respectively. It is because there were more GC operations in TRIM enabled FTL than TRIM disabled FTL. Exploiting TRIM command in FTL is essential in enjoying higher throughput; however, it calls for better design and implementation of GC scheme to be exploited along side of TRIM command.

5.5 Analysis

The major findings in this paper are two-fold. First is that FTL can exploit deallocation information from file system, which gives advantage of reducing the number of merge and GC. Second observation, on the other hand, is not so favorable, indeed it needs much attention. From our experiments, we find that TRIM in file system causes ill behavior in SSD. The side-effect comes from file system trying to exploit TRIM, while not knowing that the FTL is actively endeavoring to reduce the number of erase behind-the-scenes. When Windows 7 file system finds that the underlying SSD is TRIM enabled, Windows 7 file system allocates new file system block to perform update instead of overwriting the new content to the same logical block. Similar behavior can be found in log-structured file system[12]. Since the file system always performs 'write' to new block which is free block in SSD, this approach entails less merge operation for Page FTL. However, since this file system always writes to free block, it quickly uses up the free blocks in the SSD and garbage collection is triggered in relatively frequent manner. The overhead of garbage collection is orders of magnitude larger than that of merge operation. In our experiment, OS installation actually takes 5.7 times longer in Windows 7 when we use TRIM enabled SSD with hybrid FTL. This shows that although TRIM command brings good news to SSD, it still needs proper and elegant support from the file system to reduce the conflict and to achieve the better performance.

6. CONCLUSIONS

Recent introduction of TRIM command to T13 Technical Committee that allows a file system to inform locations of erased sectors to SSD has opened possibilities to reduce the cost of merge operation. However, FTL designers are responsible for implementing the proposed interface. In this paper, we propose data structure to manage the deallocation information and search the information. We also propose how and when to exploit this information in the FTL under multi-channel architecture. We measure effectiveness of TRIM enabled FTLs, Page Mapping FTL and Hybrid Mapping FTL, in real time system emulator with system-monitoring tool. Performance of TRIM enabled FTL compared to FTL without TRIM capability shows 2% and 13% increase in Hybrid and Page Mapping FTL, respectively. Although exploiting TRIM in Page Mapping FTL provides 10.7% better throughput in random write IOZone benchmark, Page Mapping FTL can suffer from GC as the storage utilization becomes high and available storage area becomes low, which requires careful design and implementation of GC algorithm.

7. REFERENCES

[1] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. Manasse, and R. Panigrahy. Design tradeoffs for ssd performance. In *ATC'08: USENIX 2008 Annual Technical Conference on Annual Technical Conference*, pages 57–70, Berkeley, CA, USA, 2008. USENIX Association.

[2] A. Ban. Flash File System, US Patent 5,404,485, 1995.

[3] F. Bellard. Qemu, a fast and portable dynamic translator. In *ATEC '05: Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 41–41, Berkeley, CA, USA, 2005. USENIX Association.

[4] X.-Y. Hu, E. Eleftheriou, R. Haas, I. Iliadis, and R. Pletka. Write amplification analysis in flash-based solid state drives. In *SYSTOR '09: Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference*, pages 1–9, New York, NY, USA, 2009. ACM.

[5] J.-U. Kang, H. Jo, J.-S. Kim, and J. Lee. A superblock-based flash translation layer for nand flash memory. In *EMSOFT '06: Proceedings of the 6th ACM & IEEE International conference on Embedded software*, pages 161–170, New York, NY, USA, 2006. ACM.

[6] J.-U. Kang, J.-S. Kim, C. Park, H. Park, and J. Lee. A multi-channel architecture for high-performance nand flash-based storage system. *Journal of Systems Architecture*, 53(9):644 – 658, 2007.

[7] J. Kim, J. M. Kim, S. Noh, S. L. Min, and Y. Cho. A space-efficient flash translation layer for compactflash systems. *Consumer Electronics, IEEE Transactions on*, 48(2):366 –375, may 2002.

[8] S. Lee, D. Shin, Y.-J. Kim, and J. Kim. Last: locality-aware sector translation for nand flash memory-based storage systems. *SIGOPS Oper. Syst. Rev.*, 42(6):36–42, 2008.

[9] S.-W. Lee, D.-J. Park, T.-S. Chung, D.-H. Lee, S. Park, and H.-J. Song. A log buffer-based flash translation layer using fully-associative sector translation. *ACM Trans. Embed. Comput. Syst.*, 6(3):18, 2007.

[10] W. Norcott and D. Capps. Iozone filesystem benchmark. *URL: www. iozone. org*.

[11] S.-H. Park, S.-H. Ha, K. Bang, and E.-Y. Chung. Design and analysis of flash translation layers for multi-channel nand flash-based storage devices. *Consumer Electronics, IEEE Transactions on*, 55(3):1392 –1400, august 2009.

[12] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. *ACM Trans. Comput. Syst.*, 10(1):26–52, 1992.

[13] SAMSUNG Electronics. Datasheet: 2G x 8 Bit / 4G x 8 Bit NAND Flash Memory (K9XXG08UXM) , June 2006.

[14] F. Shu. Data Set Management Commands Proposal for ATA8-ACS2. *T13 Technical Committee, United States: At Attachment:e07154r1*, 2007.