A Group Round Robin Based B-tree Index Storage Scheme for Flash Memory Devices

Rize Jin
Department of Computer Engineering
Ajou University
Suwon, Republic of Korea
jinrize@ajou.ac.kr

Hyung-Ju Cho
Department of Computer Engineering
Ajou University
Suwon, Republic of Korea
hjcho@ajou.ac.kr

Tae-Sun Chung
Department of Computer Engineering
Ajou University
Suwon, Republic of Korea
tschung@ajou.ac.kr

ABSTRACT
Flash memory is rapidly deployed as data storage for embedded and tablet PCs due to its shock resistance, fast access, and low power consumption. However, it has some intractable characteristics such as erase-before-write, asymmetric read/write/erase speed, and limited number of write/erase cycles. Due to these hardware limitations, the magnetic disk-based systems and applications could hardly make full use of the advantages of flash memory when directly adopting themselves on it. For example, the frequent changes of B-tree can degrade the storage performance of flash memory. Most of the recent studies on flash-aware index design focused mainly on the buffer management scheme whereby they can reduce the costly write operations to flash. However, in this paper, we present a novel B-tree storage scheme, a group round robin based B-tree index storage scheme, which applies a dynamic grouping and round robin techniques for erase-minimized storage of B-tree in flash memory under heavy-update workload. Experiment results show that the proposed scheme is efficient for frequently changed B-tree structure and improves the I/O performance by 2.14X at best, compared to the related work.

Categories and Subject Descriptors
D.4.2 [**Operating Systems**]: Storage Management – *allocation/deallocation strategies, garbage collection, secondary storage.*

General Terms
Algorithms, Management, Performance, Design, Experimentation.

Keywords
flash memory, storage manager, dynamic grouping, round robin.

1. INTRODUCTION
Flash memory [1] shows superiority in terms of high random access speeds, low power consumption, shock/temperature resistance, small size and light weight. Therefore it has been used in a wide spectrum of computing devices, such as embedded sensors, digital cameras, mobile phones, tablet PCs, and even personal computers and enterprise servers. However, compared to traditional storage devices, flash meicmory possesses many distinguished electronic limitations: 1. erase-before-write, 2. discrepancy on read/write and erase units, 3. asymmetric read (80μs)/write (200μs)/erase (1.5ms) speed [1], and 4. a limited number of write/erase cycles. Specifically, 1. flash memory does not allow overwriting of previous data before an erase operation, 2. read and write operations are performed on a page basis, but erase operation can only be performed on a unit of a block. 3. the cost of an erase operation is much higher than read or write operation: as a result, frequent erase operation can drastically degrade the overall performance of the flash memory, 4. furthermore, each block has a limited number of erase cycles so that when certain blocks are overly used, they cannot guarantee the integrity of data [2], [3]. These make the existing systems perform poorly with a flash-based storage [2], [4]. For instance, the small and frequent random accesses issued by the upper layer (e.g., B-tree index) can cause a substantial amount of block erases and garbage collections to flash memory. Therefore, it is necessary to redesign the system software. One of the major parts of the system design that have to be revisited is the storage access design and especially the flash-based index file storage design, as an index tends to rewrite its existing files. Moreover, it becomes complicated under special circumstances, such as splitting, merging and rotating and it can propagate all the way from leaf files to the root file. Recently, many flash-based storage management schemes [2]-[4] and flash-aware index structures [5]-[12] have been proposed, separately. They are designed around a straightforward method, the sequential logging storage scheme, which concentrated on how to reducing the number of random writes to flash, relying on oversimplified buffer management scheme. In this paper, we suggest several techniques in order to apply the most-watched storage scheme, IPL [4], to the B-tree index files. Our contributions are summarized as follows:

• We derived a sophisticated approach to efficiently store and manage the B-tree on flash memory.

• We use dynamic grouping and round robin techniques to efficiently manage the number of log pages within a flash block. This contributes to reduce the number of flash erases that caused by frequently changed B-tree structure.

• We adopted the In-page Logging scheme for the index such that an index file and its log records are co-located in the same physical flash block.

• We introduced a semi-clean state to futher classify dirty pages into clean and dirty parts. This contributes to improve the buffer

IMCOM (ICUIMC)'14, January 9–11, 2014, Siem Reap, Cambodia.

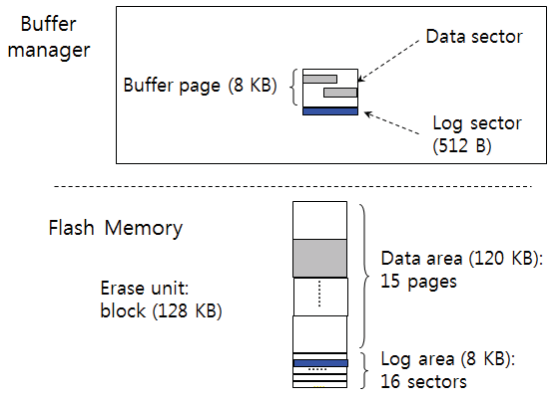

Figure 1. In-page logging storage scheme

hit ratio and space utilization of previous clean-first replacement algorithm [14].

• We conduct extensive experiments to demonstrate that the proposed storage scheme markedly outperforms the related work under heavy-update workload in terms of execution time, buffer hit ratio.

The rest of this paper is organized as follow. Section 2 discusses the background and motivation of this work. Section 3 describes the proposed solution. A group round robin based B-tree index storage scheme and its operations will be majorly presented in detail. Section 4 reports the performance evaluation. Finally, the conclusion is made in Section 5.

2. BACKGROUND

2.1 Characteristics of Flash Memory

A flash memory [1] is organized into a certain number of blocks, and each block consists of multiple pages. The internals of a flash memory differ in almost every aspect from a magnetic disk. Unlike traditional storages perform equalizing speed of read and write, the write and erase operations on flash memory take much longer time than a read operation. Reading and writing are performed on a unit of a page, and erasure can only be performed on the unit of a block. In addition, flash memory has no ability to overwrite. So, an update on a used page must be done after erasing the corresponding block. Another limitation is that flash memory has a limited number of erase cycles per block which is commonly between 100,000 to 1,000,000.

In order to overcome these electronic limitations, flash memory is supported by an intermediate software layer called Flash Translation Layer (FTL) [2], [3]. FTL helps applications above the file system to regard flash memory as a magnetic disk-like block device. Considering the difference between the basis units of write (a page) and erase (a block, consists of a number of pages), updating the outdate data on flash memory immediately is not worthwhile. FTL adopts the wear-leveling and techniques out-of-place update [2]-4]. The wear-leveling technique attempts to work around the erase cycle limitation by arranging data such that erasures are distributed evenly across the blocks. The out-of-place update technique writes the new contents of a page to another location instead of updating the page itself. As a result, the number of overwrites can be reduced. But, frequent Out-of-place update can consume the blocks and run out them. When free blocks are not enough, it has to erase the blocks who got the outdate data inside and also merge the valid data into new blocks.

2.2 In-Page Logging Scheme [4]

The in-page logging (IPL) scheme was proposed to reduce the log block thrashing and high block associativity problems of the previous log block-based FTLs [2], [3]. As shown in Figure 1, IPL divides one physical block into data pages and log pages. It writes new data for a certain data page into a log page within its block. That is, instead of writing the data page in its entirety, IPL only logs the change information (a log record), which can reduce space requirements. When a data page is to be read into the buffer, the IPL storage manager returns a copy of the data page along with its log records. Then, an in-memory procedure, the log applier, creates the up-to-date version by merging the log records with the data. IPL gains both block merge and lookup efficiencies as it stores the data and its logs in one physical block.

2.3 Flash-based B-tree Implementation

BFTL [5] is a software module for implementing/storing a B-tree on flash memory through the log block-based FTLs. It uses an in-memory write buffer to represent the requests. In its commit policy, all buffered index entries are packed into one page of flash memory. As a consequence, the number of page writes/updates is reduced. But, it equally hurts the search performance a lot, as its key entries are scattered in many different physical pages in flash memory. Some works [8, 9] do not rely on the existing FTLs and the schemes themselves provide the role of storage manager and thus can control the internals of flash storage. The IPL B^+-tree [9] reduces the number of overwrites in traditional B-tree relying on taking advantage of the IPL [4] storage access scheme.Besides, manh works [8, 11, 13] adopt the ordinary B-tree approach which simply stores the index entries in a consecutive physical area, and try to reduce the flash I/Os by exploring the index buffer management scheme. IBSF [13] eliminates the redundant index entries and then, delays the time that the index buffer requires to become full. The attractiveness of these mechanisms lies primarily in their lookup-efficient; they do not require any additional read operation for log data. But, they can't handle frequent changes made to index efficiently, and they actually failed to design a flash-aware buffer scheme.

2.4 Problem of Storing a B-tree on Flash

Consider in the naïve implementation of B-tree on flash memory, if an update occurs on a key entry in the leaf node, the updated node should be written at an empty page or a free block due to the erase-before-write limitation. Since the physical address has been changed, the pointer of its parent node also should be changed. Consequently, since the all nodes from the leaf node to root node have to be changed. The rewriting occurs many times and this results in many energy consuming write, erase and garbage collection operations [2], [4]. Even if we stored B-index files on flash memory with the most-watched storage scheme, IPL [4], the write performance of the index structure would be worse than that of magnetic disk, since the writes of the B-tree is usually small and random. An update operation occurs on a key entry in the leaf node on a data page. The changes should be written to the log page within that block to avoid erasing the whole block. In addition, the B-tree data pages are more complicated than other pages (e.g., music and document files) since, under special circumstances, the B-tree splits, merges and redistributions and this can propagate all the way from leaf node/page to the root node/page. As a consequence, the log pages will be quickly run

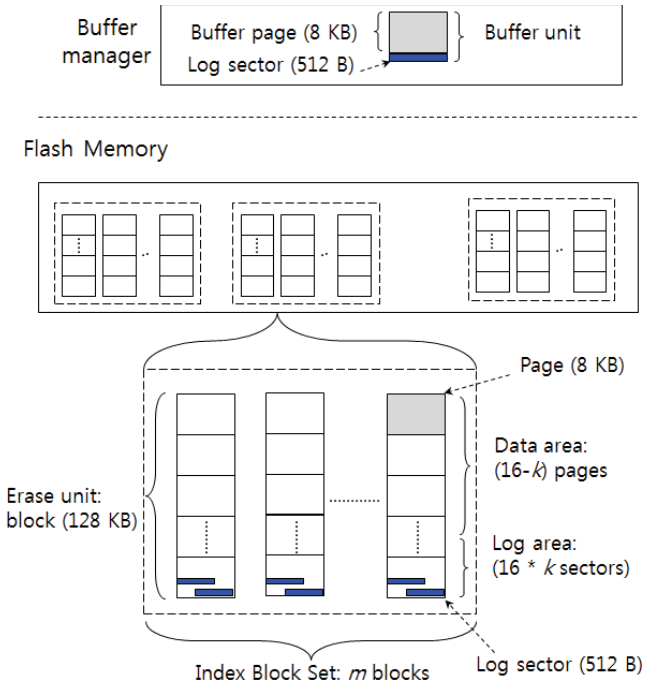

Figure 2. A group round robin based storage scheme

out and the blocks will be frequently erased under IPL's one-log-page-per-block scheme. These intensive write/erase operations can not only degrade the system performance but also shorten the lifespan of flash device. An alternative solution is to simply increase the number of log pages within a block. However, in this paper, we seek to derive a more sophisticated approach to efficiently store and manage the B-tree on flash memory.

3. GRR (GROUP ROUND ROBIN BASED STORAGE SCHEME)

For the peculiar characteristics of flash memory, the storage scheme for B-tree index should has different properties in terms of data structure and operations. However, all of the previous works use traditional sequential logging storage scheme and oversimplified buffer for storing B-tree structured files on flash memory. In this section, we propose a group round robin based B-tree index storage scheme, which efficiently eliminates the impact that caused by frequently changed B-tree structure by using dynamic grouping and round robin techniques. In addition, GRR relies on an enhanced clean-first buffer manager [14] which not only considers imbalance of read and write speeds but also the buffer hit ratio and space utilization.

3.1 System Architecture

Figure 2 shows the whole architecture of the proposed solution. And Table 1 summarizes the notations used throughout this paper. As shown, the proposed GRR divides an index block (*iB*) into data and log areas. Different from IPL, GRR groups multiple *iB* as an index block set (*iG*). In addition, GRR introduces two parameters: the number of *iB* per *iG*, *m*, and the number of log pages per index block, *k*. GRR increases or decreases the values of *m* and *k* as workload changes. IPL uses a sequential logging strategy, whereas, GRR adopts a round robin logging within *iG*.

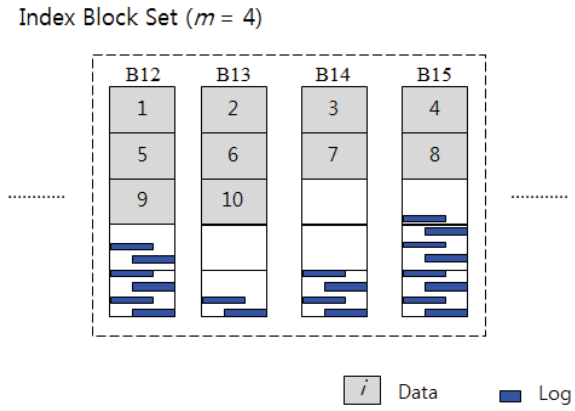

Figure 3. The round robin storing strategy

Table 1. Parameters

Symbol	Description
iB	Index block: the block that stores the B-tree nodes
iG	Index block set: consists of multiple index blocks
m	The number of *iB* per *iG*
k	The number of log pages per *iB*
l	A upper limit for *k*

3.2 Dynamic Block Grouping

When grouping a certain number of *iB* into one *iG*, GRR considers the write workload and the size of table or data file on which the B-tree is built. Specifically, *m* increases as the write workload and file size increase. Within *iB* the data area grows from top to bottom, whereas, the log area grows from bottom to top. In other words, the value of *k* increases as the number of log records increases. For efficiently merging the logs and the data, GRR gives a limit, *l*, to *k*.

3.3 Round Robin Logging

The previous studies [4], [9] write the flash blocks sequentially. However, GRR adopts a round robin writing scheme for storing the data within *iG*. Figure 3 shows an example. Numbers on the rectangles indicate the order in which they are created. As seen, the index node 1 (N1) is stored on the first page of the block 1 (B1), and the following node 2 is stored on the first page of the B2. The logs to N1 are stored only on B1. In this way, GRR not only takes the advantage of IPL scheme, but also distributes incoming write requests into multiple blocks so as to preventing wear out a certain blocks. In addition, this allows increasing the value of *l* dynamically and efficiently. Extended log area can cope with the frequent changes of B-tree. And the number of erases to flash block can be reduced.

3.4 An Enhanced Clean-First Buffer Scheme

In this paper, we introduce a semi-clean state for buffer unit to improve the buffer hit ratio and space utilization of CFLRU. In GRR, a cached index node/page can have three states, which are defined as follows:

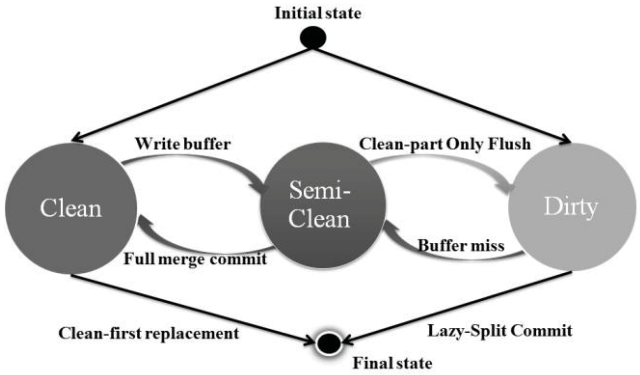

Figure 4. State transition diagram of the buffer pages under three-state scheme

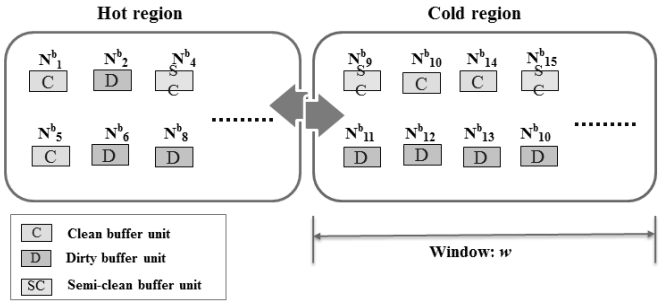

Figure 5. An example of the enhanced clean-first LRU buffer management

Clean State: If a buffer unit does not contain any changes/updates to the corresponding flash page then it is said to be a clean buffer page.

Semi-clean State: If a buffer unit consists of the changes/updates and the original data of the corresponding flash page then it is said to be a semi-clean buffer page.

Dirty State: If a buffer unit only contains the changes/updates to the corresponding flash page then it is said to be a dirty buffer page.

The initial state of a buffer unit can either be clean state or dirty state. After updating, a clean state buffer unit transitions to semi-clean state, instead of to the dirty state. The semi-clean state page can only transition to the dirty state by flushing out the log records of its due to the clean-first strategy. Figure 4 describes the state transitions of a buffer unit under the three-state scheme.

Figure 5 shows an example of the enhanced CFLRU scheme. The LRU list is divided into two regions, namely the hot region and the cold region. The *hot region* consists of recently used pages and most of the cache hits are expected to be generated in this region. The *cold region* consists of pages that are candidates for eviction. The clean state pages or the clean parts of semi-clean state pages are preferentially selected as victims over dirty state pages in the cold region. To do this, we keep two lists for the pages in the cold region: if a node/page is in the clean state or semi-clean state then it goes to the *clean list*, otherwise it is listed on the *dirty list*. Upon a buffer miss, if the clean list is not empty, then the tail of the clean list is returned as the victim. Otherwise, some dirty pages will be evicted using the *modify-two-node*/*lazy-split commit policy* [15]. As the buffer miss ratio may increase if the recently used clean page is evicted, it is important to properly adjust the size of the cold region (called a window size, *w*). Here, only the pages within *w* can be candidate victims.

3.5 Operations

There are two types of operations required for GRR. The first type is a read operation, which is requested when a buffer miss occurs and have to read an index node from flash memory. The other one is a write operation, which is performed to flush log records into flash memory.

3.5.1 Read operation

When a search operation is issued for a key *K*, starting at the root, GRR is traversed from top to bottom. At each level, the search chooses the child node that satisfies the search condition, and if that node can be found in the buffer pool, its address is returned. Otherwise, the node and its log data are loaded from *iB* to construct the up-to-date version as a buffer unit. The buffer manager allocates a buffer frame for the unit and returns its address.

3.5.2 Write operation

Write operations of GRR can be performed to append the log records to the corresponding in-memory log sector, as shown in Figure 2. When a buffer unit becomes full and needs to be flushed, the buffer manager discards clean pages preferentially as long as the buffer hit ratio is not badly harmed. And when we have to evict the dirty buffer units, we just flush those that might cause fewer node splits. IPL checks whether the corresponding flash block can accommodate the log records of the dirty part. If it does, the log records are flushed to flash memory. Otherwise, it allocates a free block and constructs the up-to-date version of each index node stored in the data pages by applying their logs. In contrast, GRR dynamically increases the number of log pages, just avoiding frequently erasing data blocks and consuming free blocks.

4. PERFORMANCE EVALUATION

To evaluate the effects of the proposed storage scheme, we compare the numbers of page copies and block erases under IPL and GRR schemes and also give the effect of the proposed concept of semi-clean state. The result is summarized here.

4.1 Cost Model

B-tree index pages more venerable than other data pages and generate many logs which can wear down the log area of IPL quickly and increase the number of block erase operations. In contrast, the proposed scheme dynamically enlarges the log area and the size of index block set. This contributes to the reduction of block erase count and the wear-leveling on flash memory. We except that in the most case the index block set can accommodate

Table 2. Parameters Cost Analysis

	Cost Model
Erase	$0 \leq Erase(GRR) \leq Erase(IPL)$
Read	$2*Read(Flash) = Read(IPL) \leq Read(GRR) \leq (+1) Read(Flash)$
Space	$Space(IPL) \leq Space(GRR)$

Table 3. Access Features

Flash Memory	Page Read	Random Access	80 μs (Max.)
		Serial Page Access	50 ns (Min.)
	Page Write	Program Time	200 μs (Typ.)
		Block Erase Time	1.5 ms (Typ.)

Table 4. Specifications of Dataset

Para.	Description	Value
nfc	# of file creation	{1K, ***5K***, 10K}
m	the number of *iB* per *iG*	{***2, 4, 8, 16***}
k	the number of log pages per *iB*	{***1, 2, 4, 8***}
fmc	file modification coefficient	{0, 1, 2, ***4***, 8}
w	window size	{0, 0.25, 0.5, ***0.75***, 1}
WR	amount of logical pages reached by write requests	***1024000***
AWL	average write/update length	***16 K***
RWR	read/write ratio	{100/0, 80/20, 60/40, 40/60, ***20/80***, 0/100}
ARL	average read length (pages)	***1***
l	A upper limit for *k*	***8***

all the data and logs and incur no erase operation. Table 2 compares the costs of erase and read operations and the space utilizations of IPL and GRR. Though the number of reads and the utilized space can be increased in GRR, there is a considerable decrease to the number of erase operations, and just achieving a better overall performance.

4.2 Experiment Settings

The simulation studies are performed on a Core i5-2500 machine (3.30GHz, 3GB DDR2 RAM) which is running Ubuntu 10.10 operating system (Linux 2.6.35). In the simulation, we assume that the embedded system has 128 MB SDRAM and a 4 GB flash memory, and we use the weighted costs of read, write, and erase operations, as shown in Table 3.

The proposed solution is tested under many different settings. Table 4 provides details of the experiment set. The typical settings we choose for our evaluations are presented in bold italics. The *fmc* represents the average number of modifications that are made on each file data. For example, if *fmc* is 4, a single file is said to have been modified four times. The *w* represents the proportion of the cold region to the total buffer space.

4.3 Varying *m* and *k*

Figure 6 compares the performances of IPL and GRR under a random trace. We varied *k* with *m* values fixed as 8 and then varied *m* with a fixed *k* of 4, as shown on the X-axis. The Y-axis shows the total execution times, In the case of increasing the values of *k*, GRR shows the best performance when *k* is 4, since the bigger *k*, the heavier the merge work. Whereas, for the case of increasing the values of *m*, the performance grows as *m* increases.

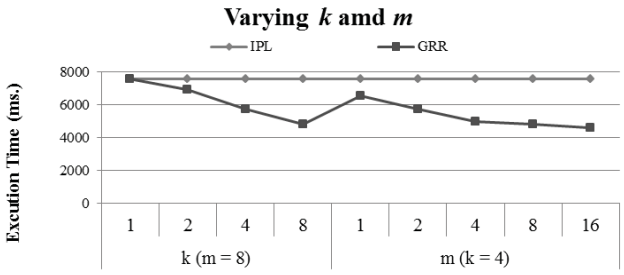

Figure 6. Comparisons of execution times by varing *k* and *m* under random access trace)

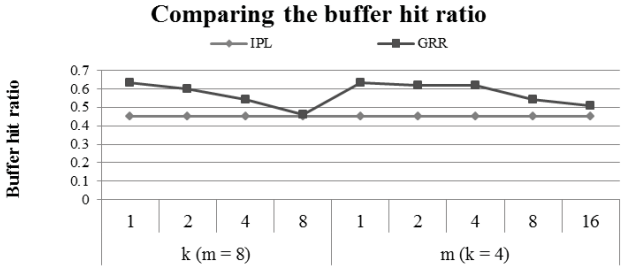

Figure 7. Comparisons of the buffer hit ratios

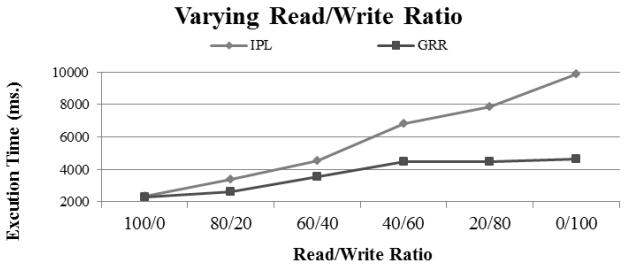

Figure 8. Effects of increasing random write operation ratio

4.4 Buffer Hit Ratio

We also counted the buffer references that were measured by the above test. Since increasing the values of *k* or *m*, will decrease the number of clean pages or clean part of semi-clean pages, and just lower the buffer hit ratio. As shown in Figure 7, when *k* is 8 or *m* is 16, GRR and IPL have similar buffer hit ratios. However, GRR always perform better than IPL.

4.5 Varying Read/Write Ratio

Finally, we measured the performances of GRR and IPL with *l* is 8of 4 and the value of *m* values fixed at 8 by varing *RWR*. The results shown in Figure 8. Under a heavy-read workload, GRR and IPL have the same effects, since the most of the cached pages are clean pages. However, the performance gaps between them increase as the write ratio increases. It is clear that IPL is not optimal with a heavy-write workload for a B-tree files. Compared to IPL, GRR is expected to perform fewer writes to flash memory since it adopts an enhanced clean-first replacement policy GRR clearly outperforms IPL when the write ratio is larger than 60%, mostly because it uses more log pages for one block to reduce the number of flash erases. Under a ratio of 0/100, the effect of GRR is large, and the access performance of it is improved by 2.14X compared to IPL.

5. CONCLUSIONS

Despite the many advantages of flash memory, existing storage schemes perform poorly using it because of its unique electronic limitations. Especially, the B-tree index file more venerable than other data pages and generate many logs which can wear down the log area of IPL, the most-watched FTL scheme, quickly and increase the number of block erase operations. In this paper, we proposed a group round robin based B-tree index storage scheme, GRR, which uses the dynamic grouping, round robin techniques and an enhanced clean-first buffer replacement policy. The trace driven simulation showed that GRR efficiently eliminates the number of block erases that caused by frequently changed B-tree structure. In detail, when compared with the IPL, GRR increases the buffer hit ratio at most by 1.60X, and improves the performance at best by 1.67X under a random workload, and by 2.14X under a update-intensive workload. Additionally, we expect that GRR could easily be adopted by other types of files, such as vedio, metadata files, and that this kind of work should be addressed in the future. In addition, we intend to study a dynamic adaptation technique that adjusts the size of log area, window size, and index block set, etc..

6. ACKNOWLEDGMENTS

This work was partially supported by Basic Science Research Programs through the National Research Foundation of Korea funded by the Ministry of Education, Science and Technology (2013R1A1A2A10012956 and 2012R1A1A2043422).

7. REFERENCES

[1] Samsung Electronics. 2009. NAND based solid state drive. *SSD (MMDPE56G8DXP-0VB) data sheet*, 2009.

[2] Tae-Sung Chung, Dong-Joo Park, Sangwon Park, et al. A survey of Flash Translation Layer. *Journal of Systems Architecture Embedded Systems Design*, *55* (2009), 332-343.

[3] Sang-Won Lee, Dong-joo Park, Tae-Sun Chung, Dong-Ho Lee, Sangwon Park, and Ha-Joo Song. 2007. A log buffer-based flash translation layer using fully-associative sector translation. *ACM Trans. on Embedded Comp. Syst.*, *6* (2007).

[4] Sang-Won Lee and Bongki Moon. 2007. Design of flash-based dbms: An in-page logging approach. In *Proceedings of the ACM SIGMOD*. 2007, 55–66

[5] Chin-Hsien Wu, and Tei-Wei Kuo, Li-Pin Chang. 2007. An Efficient B-Tree Layer Implementation for Flash-Memory Storage Systems. *ACM Transactions on Embedded Computing Systems*, *6*, 3 (July 2007).

[6] Nath, S., Kansal, A. 2007. FlashDB: Dynamic self-tuning database for NAND flash. In *Proceedings of the 6th International conference on Information processing in sensor networks* (Cambridge, MA, USA). IPSN '07, 410-419, 2007.

[7] Devesh Agrawal, Deepak Ganesan, Ramesh K. Sitaraman, Yanlei Diao, Shashi Singh. 2009. Lazy-Adaptive Tree: An Optimized Index Structure for Flash Devices. *VLDB Endow*, *2*, 1 (2009), 361-372.

[8] D. Kang, D. Jung, J. U. Kang, and J. S. Kim. μ-tree: An ordered index structure for NAND flash memory. In *Proceedings of ACM Conference on Embedded Systems Software*, 144-153, 2007.

[9] Gap-Joo Na, Sang-Won Lee, Bongki Moon. Dynamic In-Page Logging for Flash-Aware B-Tree Index. In *Proceedings of the 18th ACM Conference on Information and Knowledge Management* (Hong Kong, China), 1485-1488. 2009.

[10] Suman Nath, Phillip B. Gibbons. Online maintenance of very large random samples on flash storage. *VLDB Journal, 19*, 1 (2010), 67-90.

[11] Yinan Li, Bingsheng He, Robin Jun Yang, Qiong Luo, Ke Yi. Tree Indexing on Solid State Drives. *VLDB Endowment*, *3* (2010).

[12] Martin V. Jørgensen, René B. Rasmussen, Simonas Šaltenis, Carsten Schjønning. FB-Tree: A B^+-Tree for Flash-Based SSDs. In *Proceedings of the 15th Symposium on International Database Engineering & Applications (IDEAS '11)* (Lisbon, Portugal, 2011). 34-42.

[13] Hyun-Seob Lee, Dong-Ho Lee. An efficient index buffer management scheme for implementing a B-tree on NAND flash memory. *Data Knowl. Eng.*, *69*, 9 (2010), 901-916.

[14] Seon-yeong Park, Dawoon Jung, et al. 2006. CFLRU: a replacement algorithm for flash memory. In *Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems (CASES '06)* (Seoul, Korea, October 23-25, 2006). 234–241

[15] Rize Jin, Se Jin Kwon, Tae-Sun Chung. 2011. FlashB-tree: a novel B-tree index scheme for solid state drives. In *Proceedings of the 2011 ACM Symposium on Research in Applied Computation* (Miami, FL, USA, November 2-5, 2011). 50-55.

[16] X. Tang, X. Meng. ACR: an Adaptive Cost-Aware Buffer Replacement Algorithm for Flash Storage Devices. In *Proceedings of the International Conference on Mobile Data Management (MDM '10) (*Kansas City, Missouri, USA, 2010).