International Journal of Internet, Broadcasting and Communication Vol.14 No.3 161-170 (2022)
http://dx.doi.org/10.7236/IJIBC.2022.14.3.161

IJIBC 22-3-20

A New Flash-aware Buffering Scheme Supporting Virtual Page Flushing

Seong-Chae Lim

Professor, Dept. of Computer Science, Dongduk Women's University, Korea
sclim@dongduk.ac.kr

Abstract

Recently, NAND-type flash memory has been regarded to be new promising storage media for large-scale database systems. For flash memory to be employed for that purpose, we need to reduce its expensive update cost caused by the inablity of in-place updates. To remedy such a drawback in flash memory, we propose a new flash-aware buffering scheme that enables virtual flushing of dirty pages. To this end, we slightly alter the tradional algorithms used for the logging scheme and buffer management scheme. By using the mechanism of virtual flushing, our proposed buffering scheme can efficiently prevent the frequenct occureces of page updates in flash storage. Besides the advantage of reduced page updates, the proposed viurtual flushing mechanism works favorably for shorneing a recocery time in the presense of failure. This is because it can reduce the time for redo actions during a recovry process. Owing to those two benefits, we can say that our scheme couble be very profitable when it is incorporated into cutting-edge flash-based database systems.

Keywords: *flash memory, buffering scheme, recovery, database system.*

1. Introduction

Over the past decades, the NAND-type flash memory has drawn much attention as new storage media [1-11]. As flash's price per bit is getting cheaper at a fast rate, database communities have regarded flash memory as main storage media of high-performance database systems [1-3]. For this reason, many researchers proposed a variety of ways to utilize the benefits of fast random read/write speeds of flash, thereby improving the performance of database systems. In these researches, a major technical challenge tacked is the poor update performance of flash storage, which is caused by its inability of in-place updates [2-5, 8-11]. We also focus on that problem found in flash-based database systems.

In the case of traditional database systems, it is common to use an in-memory buffer pool for better I/O throughput [3, 6-8]. The buffer pool has a set of fixed-size memory frames for caching hot/warm data pages in there. By accessing and manipulating data pages cached in the buffer pool, a database system can reduce a large fraction of physical (or actual) I/O's in its storage, which traditionally consists of HDDs (Hard Disk Drives).

While a set of data pages is existing in the buffer pool, flushing of a dirty page occurs in two cases: (i) when the dirty page is evicted according to a buffer replacement algorithm; or (ii) when a log manager decides to write it for the purpose of fast recovery [8, 11, 12]. Note that flushing means the occurrence of a page update. The former case is for reclaiming free space in the buffer pool; the latter case is relevant with the logging mechanism of database systems. Let us look into the latter case in more detail. For fast recovery after system failure, the database system employs a log manager responsible for storing log records and a dirty page table.

Manuscript Received: June. 24, 2022 / Revised: June. 27, 2022 / Accepted: July. 2, 2022
Corresponding Author: sclim@dongduk.ac.kr
Tel: +82-02-940-4589, Fax: +82-02-940-4170
Professor, Dept. of Computer Science, Dongduk Women's University, Korea

The log manager saves log records in accordance with the well-known WAL (write-ahead logging) protocol [12-17]. As one of WAL's advantages, the buffer pool can get a NO-FORCE policy, thereby omitting expensive (physical) flushing of dirty pages at the points of every transaction commit. Although the NO-FORCE policy contributes to less I/O workload, delayed update times of dirty pages inevitably increase times for recovering from system failure [15, 17]. To solve the prolonged recovery time, the database log manager periodically flushes *aged* dirty pages regardless of availability of free buffer pages. Such periodic flushing is the cause of the latter case of flushing that we stated before.

To reduce physical flushing, that is, updating of pages, we devise a new buffering scheme that can replace flushing of a dirty pages with a pure-write and a page read. Here, a *pure-write* is an I/O operation used for writing data into an empty page. Note that in flash a page update needs a pure-write and invalidation of an old page because of flash's inability of the in-place update. To enable such replacing between I/O operations, we alter a traditional buffer management table, thereby tracing the existence of flushing having no real physical update. In the paper, we refer to such update-free flushing of a dirty page as **virtual flushing**.

To see how to execute virtual flushing, suppose that a buffer replacement algorithm has chosen page *P* as a victim page. If virtual flushing is executed for page *P*, then that page will be evicted from the buffer pool without its updating. When page *P* is again requested for access, our proposed buffer pool manager will do a redo-like action for restoring the up-to-date image of *P* in the buffer pool. For this purpose, our buffer pool manager saves a log record just before the eviction time of page *P*. Since the log size is tiny and its saving is done through a pure-write, we can reduce the workloads of update operations by utilizing virtual flushing. Since the use of virtual flushing helps to reduce the number of updates on a database, it works favorably for shortening the recovery time with less logging overhead.

The organization of this paper is as follows. In Section 2, we introduce I/O characteristics of flash memory and give an overview for the proposed buffering scheme. In Section 3, we present the proposed algorithms for a flash-aware buffer pool in detail. In Section 4, we estimate the performance improvement based on a mathematical I/O model, and then conclude this paper in Section 5
re

2. Preliminaries

To address the proposed flash-aware buffering scheme, we first go over some characteristic I/O features of flash memory. Next, we explain the underlying ideas of the proposed buffering scheme.

2.1. Previous Works

Because of advantages of very fast random I/O's, flash memory seems to be a promising media for main storage of database systems in the near future [1-6]. To render this realistic, many researchers attempted to solve the problem of the asymmetrically high cost of update operations in flash storage. Note that such asymmetricity of update costs between other I/O operations is not true in the case of HDD storage. Since an in-place update is not possible for flash memory, flash relies on an extra software module called FTL (Flash Translation Layer) [12, 16-18]. To simulate in-place updates in flash storage, FTL constantly manipulates internal metadata so as to locate a correct disk address with respect to every logical page ID. When consecutive out-of-place updates exhaust empty pages, FTL performs actions for garbage collection. Due to very high overheads for garbage collection, an update operation becomes very expensive in flash storage [12, 15, 18]

For this reason, much literature on flash-based database systems is aimed at diminishing the frequency of update operations arising in the database system. Such literature can be classified into two categories, that is, logging based approaches [8, 14-16] and caching based approaches [5, 6, 9-11]. The former approach prepares a log area in each flash block and saves log records of update operations, rather than updating pages. When an updated page is retrieved later, its correct latest image is computed by applying a redo action on its old image. For fast redo actions, the physical redo scheme is usually accepted by saving difference between the latest image and the old image of an updated page [8, 20]. From this, this approach can replace update operations with pure-writes for logging and physical redo's.

On the other hand, the latter approach saves update histories in buffer memory to cache update operations arising on nodes of a B-tree. To save previous updates arising in the B-tree, for instance, literature saves log data of key inserts and key deletes in that memory area [7, 8, 14]. Because such a cache area is made within

internal levels of the B-tree, downward key searches is possible without heavy overheads. By referring to log data on the way to a target leaf node, a key searcher can do correctly its key search. When the memory area in the buffer becomes full of update log, the area is initialized by writing the history of key inserts/deletes downwards leaf nodes at once.

Those two approaches have some problems. In the case of the logging based approach, it easily suffers from wastes of storage space to reserve a log area in every storage block. Moreover, they have to repeatedly perform expensive garbage collection to initialize the log are in the block [18]. Because of heavy overhead, this logging based approach seems to be not feasible for high-performance database systems. On the other hand, the latter caching approach demands large extra memory used for temporarily logging updates operations. Furthermore, for its use, one has to alter a significant portion of the traditional buffering algorithm [1, 9, 13]. Because of increasing memory usage and heavy burden in code alternation, it is not easy to adopt this approach for the general-purpose database systems. To avoid those problems, we propose a flash-aware buffering scheme that does not need extra memory or significant modifications of the conventional buffering algorithms

2.2 Idea Sketch

To describe the underlying ideas of our proposed buffering scheme, we first overview the way the traditional buffer manager works with a log manager in a database system. When an I/O request is issued for updating a page *P*, the buffer manager first checks if the ID of page *P* exits in the list of pages cached in the buffer pool. If it is found there, then page *P* is modified in the buffer pool without physical I/O; otherwise, I/O for reading page *P* is needed before its update. When page *P* is updated in the buffer pool, the buffer manager sets *P*'s state to a dirty one. Note that the state of page *P* remains dirty until *P* is flushed to storage. At the same time, the log manager performs logging actions by following the WAL protocol [12, 16]. That is, every update on page *P* is logged and the associated log records are written prior to the flushing time of *P*. With the log records, a recovery process can be conducted in the presence of system failure.

The recovery procedure is comprised of three consecutive phases, that is, a log analysis phase, a redo phase, and an undo phase. Since the redo phase usually accounts for a major portion of the overall recovery time, it is crucial to shorten the time for the redo phase [12, 16]. The redo phase restores the up-to-date (or valid) images of dirty pages that are checkpointed in the DPT (dirty page table). DPT contains IDs of dirty pages and log sequence numbers (LSNs) of recovery log records associated with those dirty pages. Here, the recovery log record of a dirty page *P* denotes a log record that was create for logging the first update on *P* after *P* is buffered. If a dirty page *P* has a recovery log record *R*, then the LSN of record *R* acts as a beginning point for redoing the page. Therefore, a restarted system checks the LSNs of all the recovery records and uses the oldest one as the beginning point of the redo phase. If any dirty page resides long in the buffer pool without its flushing to storage, then it is likely to enlarge the time for redo phase. For this reason, most of database systems take an approach to flush aged dirty pages in background mode for the fast recovery process [2, 12, 16].

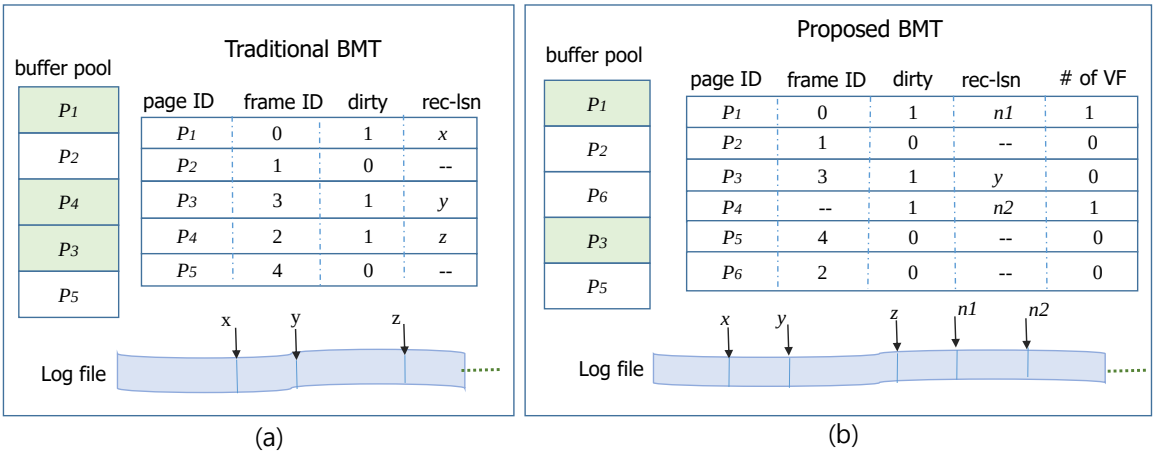

Figure 1. The data structures of the BMTs used in a conventional buffer pool and a proposed one.

To explain how a buffer manager works with a log manager, we use Figure 1. In the figure, we assume

that the buffer pool consist of five buffer frames, and the LSN of log record *R* is the same as the file offset where record *R* is saved in the log file. As depicted in Figure 1(a), five pages have been cached in the buffer pool and three pages of *P1*, *P3*, and *P4* are assumed to be dirty pages. Correspondingly, their dirty bits are set to 1 in the buffer management table (BMT). The field "**rec-lsn"** in BMT saves the LSN's of recovery log records of the dirty pages. In the figure, the recovery log record of *P1* has *x* as its LSN. If the current BMT is checkpointed and the system fails shortly, a redo phase will be performed from the log file offset of *x*. because *x* is the oldest LSN of all the recovery records.

As stated earlier, flushing of a dirty page arises when it is needed for any of buffer replacement or the fast recovery consideration. To replace that with the proposed virtual flushing, we slightly alter the data structure of BMT as well as a logging mechanism. Our proposed BMT is shown in Figure 1(b), where we assume that the buffer pool state has come from one of Figure 1(a). That is, we assume that page *P4* of Figure 1(a) had been evicted from the buffer pool and its frame was reallocated for page *P6* as in Figure 1(b). The modified BMT contains an additional field "**# of VF**", which is for counting the frequency of virtual-flushing arising for a corresponding dirty page. To count virtual-flushing of any dirty page *P*, we made increment on that field. In the case of Figure 1(b), virtual flushing has arisen on two pages of *P1* and *P4*, respectively.

To present the proposed scheme for virtual flushing, we use the case of page *P4* in Figure 1(b). Here, we assume that *P4* was evicted according to a buffer replacement algorithm, and its used buffer frame has been reallocated to page *P6*. As shown in Figure 1(b), the buffer frame of ID 2 has been allocated to page *P6*. In the case of traditional buffering scheme without virtual flushing, we need to delete page *P4* from BMT. Rather than such physical flushing, our buffering scheme just saves a redo log record representing the update on *P4*, and indicates the existence of virtual flushing by using "**# of VF**". At the same time, field "**rec-lsn**" is updated to LSN of that new log record, that is, *n2* in Figure 1(b). This kind of virtual flushing is one executed for avoiding a traditional flushing during a buffer replacement time.

Next, we present other case of virtual flushing, which is performed for the consideration of fast recovery. In the case of page *P1* in Figure 1(b), virtual flushing was executed for fast recovery. Unlike page *P4*, *P1* still remains in the buffer pool, since flushing is aimed at fast recovery. Instead flushing the current image of *P1*, a new log record is saved for *P1*. Thanks to the new log record with LSN *n1*, the beginning point of the redo phase is moved from x to y. Therefore, our buffering scheme supports fast recovery, while avoiding an update on *P1*. In the next section, we give detailed algorithms needed for the proposed virtual flushing.

3. Proposed Flash-aware Buffering Scheme

3.1 Algorithm for Virtual Flushing

For the implementation of virtual flushing, the proposed buffer manager use BTM with an additional field, named **"# of VF"**. This field is used for counting executions of virtual flushing on each dirty page. With the combinational use of this field and other traditional fields, the proposed buffer manager can differentiate between four states of buffered pages as in Figure 2.

	page ID	frame ID	dirty	rec-lsn	# of VF
state 1	*valid*	*valid*	0	*null*	*zero*
state 2	*valid*	*valid*	1	*valid*	*zero*
state 3	*valid*	*valid*	1	*valid*	*> zero*
state 4	*null*	*null*	1	*valid*	*> zero*

Figure 1. Four kinds of an in-buffer page managed by our buffering scheme.

Among those four states, two states are the same ones used in a traditional buffering scheme, that is, state 1 is for a clean page, and state 2 is for a dirty page. Other two states of 3 and 4 are given to any page having virtual flushing. In the case of a page with state 3, field "**frame ID**" has a valid frame ID, since that page

remains in the buffer pool. If page *P* has state 3, then we can say that virtual flushing has been done on *P* for fast recovery. On the other hand, a page with state 4 has *null* as the value of "**frame ID**". When the page of state 4 is accessed again, our buffer manger restores the up-to-date image of that page. Such page restoring is done by using the associated redo log record that was created at the time of virtual flushing. For fast restoring of the page of state 4, our scheme uses the physical redo record that save difference image between the old image and the new image. The way for saving compact physical redo log can be referred to earlier literature [8, 12, 16].

The algorithm for virtual flushing is given in Figure 3. When our buffer manager executes actions for page flushing, it executes procedure *FlushPageFromBuffer()*. If the procedure is invoked at the time of buffer replacement, the parameter of *cause* is set to **pageOut**; otherwise, if the procedure is called for the consideration of fast recovery, then the parameter of *cause* has the value of **fastRecovery**.

Algorithm 1: Procedure *FlushPageFromBuffer*

Input : Pid = page ID of a target page;
BMT = object for accessing the buffer management table;
$Buffer$ = object for accessing the buffer pool;
$cause$ = **fastRecovery** or **pageOut**;

```
1  e ← BMT.getEntry(Pid); // get the BMT entry made for Pid
2  if e.NumVF ≥ MaxLogging then // checking the pervious frequency of VF
3  |  bVirtualFlush ← false; // physical flushing is needed
4  else
5  |  bVirtualFlush ← true; // virtul flushing will be done
6  if bVirtualFlush = true then // virtual flushing is done
7  |  lsn ← Buffer.writeLogRec(Pid); // creation of a new log record
8  |  (e.NumVF, e.recLSN) ← (e.NumVF + 1, lsn); // change of two fields of e
9  |  if cause = pageOut then
10 |  |  Return the frame used by e.frameID to the free buffer pool;
11 |  |  e.frameID ← null; // change of the frame-ID field
12 |  BMT.updateEntry(Pid, e); // update of the associated BMT entry
13 else // physical flushing is done
14 |  Buffer.writePage(Pid); // physical flushing to storage
15 |  if cause = pageOut then // a frame is returned to the buffer pool
16 |  |  Return the frame used by e.frameID to the free buffer pool;
17 |  |  BMT.deleteEntry(Pid); // deletion of the BMT entry
18 |  else // a BMT entry is updated
19 |  |  (e.dirtyBit, e.recLSN, e.NumVF) ← (0, 0, 0);
20 |  |  BMT.updateEntry(Pid, e); // update of the associated BMT entry
```

Figure 3. Proposed buffering algorithm for flushing a dirty page.

3.2 Algorithm for Reading a Page

In this subsection, we present how to process a page read requested by an application process. The I/O request is issued through procedure *GetPageFromBuffer()* of Figure 4. In the figure, we assume that the procedure accepts page ID of a target page and returns the buffer frame address allocated to that page. For this, the procedure first checks if the target page of *Pid* exists in the list of BMT. If not found, then the procedure caches that page in the buffer pool. The steps for such buffering are described in lines 2-5.

Otherwise, if the target page is already cached one, then in line 7 the procedure checks whether or not that page exists in the buffer pool. If that page is a currently cached one, the procedure just returns the frame address allocated to the target page. Recall that the step in line 14 is for handling a page of states 1, 2 or 3 of Figure 2.

To read a target page of state 4, the procedure performs the steps of lines 8-12, thereby restoring the correct image of the target page. Since the up-to-date image of target page *P* was removed from the buffer pool, we

need to restore *P* 's correct image by applying the associated log record to *P*'s old image saved in storage. For this, both of the log record and the old image of *P* are read lines 8-9, respectively. In line 10, a physical redo is executed on *P*, thereby restoring the valid target image. Finally, the BMT entry is updated to change page's state from state 4 to state 3 in line 12.

Algorithm 2: Procedure *GetPageFromBuffer*

Input : Pid = ID of the data page to be fetched;
BMT = object for the buffer management table;
$Buffer$ = object for the buffer pool;
Return: Memory address of the frame used for page Pid;

```
1  e ← BMT.getEntry(Pid); // get the BMT entry for Pid
2  if e = null then // there is no entry for page Pid
3  |  Frame ← Buffer.read(Pid); // read of Pid from storage
4  |  e ← (Pid, Frame, 0, null, 0); // create a BMT entry
5  |  BMT.saveEntry(e); // entry of state 1
6  else
7  |  if e.frameID = null then // page of state 4
8  |  |  LogRec ← Buffer.readLog(e.rec-lsn); // from a log file
9  |  |  Frame ← Buffer.read(Pid);
10 |  |  Buffer.overwrite(Frame, LogRec); // physical redo
11 |  |  e.frameID ← Frame; // reflecting of Pid's restoring
12 |  |  BMT.updateEntry(e);
13 |  else
14 |  |  Frame ← e.frameID; // page of state 2 or 3
15 Return the memory address of Frame;
```

Figure 4. Algorithm for getting the buffer address where a target page is cached

3.3 Recovery Algorithm

Because our buffering scheme needs to save redo log records used for virtual flushing, our algorithm for the recovery process has to take into account this type of log records. More specifically, the algorithm for the redo phase is slightly modified because the proposed scheme for virtual flushing does not save any undo log records. Recall that the recovery process is comprised of via three phases, that is, a log analysis phase, a redo phase, and an undo phase [8, 12, 16]. For this reason, we focus on the algorithm needed for a redo phase.

During a log analysis phase after system failure, a recovery module reads the latest checkpoint record and builds a list of aborted transactions by scanning a log file. Then, the recovery module executes a redo phase by following the algorithm of Figure 5. For exposition brevity, we assume that the checkpointed BMT has been reincarnated during a log analysis phase.

Algorithm 3: Procedure *DoRedoPhase()*

```
Input  : BMT = object for accessing a buffer management table;
         Buffer = object for accessing a buffer pool;
         LogFile = object for accessing a log file;
1  LogFile.locate(redo_start); // set the log file pointer to LSN of the oldest
     recovery record
2  while LogFile.isEnd() ≠ true do // log scanning
3      R_log ← LogFile.getNextLog(); // read of a log record
4      if R_log is a log record for virtual flushing then
5          Perform a physical redo on the assocated page using R_log;
6  LogFile.locate(redo_start); // file relocating for traditional redo actions
7  while LogFile.isEnd() ≠ true do // log scanning
8      R_log ← LogFile.getNextLog(); // for normal redo logs
9      if BMT.isExisting(R_log.pageID) ≠true then
10         Frame ← Buffer.readPage(R_log.pageID); // buffering of this page
11     lsn ← Buffer.getLSN(Frame); // LSN of the buffered page
12     e ← BMT.getEntry(R_log.pageID); // get a BMT entry
13     if R_log.LSN > lsn then // check if redo is required on this page
14         Perform a redo action on the buffered page by using R_log;
```

Figure 5. Algorithm of *DoRedoPhase()* used for executing redo actions for recovery.

In line 1, the redo algorithm locates the log file pointer to the redo starting pointer. The starting point is determined as the least LSN of recovery log records saved in the checkpoint record. During the steps of lines 2-5, the algorithm performs physical redo's by using log records that have been saved for virtual flushing. After completing redo actions relevant with virtual flushing, the algorithm performs the redo actions again, while reading other redo log records that are not associated with virtual flushing. For this, the procedure relocates log file pointer to the redo starting point in line 6, and then works in accordance with conventional redo algorithm as in lines 7-14.

4. Performance Evaluation

Because of the absence of in-place update capability of flash memory, the performance of flash storage is apt to get worse when update operations occurs frequently. Against the problem, we have proposed a new flash-aware buffering scheme. When a dirty page needs flushing for any reason, our scheme saves a small size of a log record and properly manipulates a buffer management table. By superseding costly updates with cheap pure-writes of log data, our scheme can improve the I/O throughput of flash database systems. Especially, since it is easy to efficiently reclaim the storage space of the used log file through TRIM or UNMAP [19], our buffering scheme can fully capitalize on the advantage of the infrequency of garbage collection as well.

Although the proposed buffering scheme can reduce occurrences of real update operations, it has to pay some extra costs for restoring dirty pages with virtual flushing. Those extra costs are composed of I/O costs for reading log records and CPU times for conducting redo actions. Here, a CPU time for a redo action is taken for copying after-images of a page into the buffer. Since the cost of a memory copy is very cheap compared with other I/O costs, we neglect the memory copy cost in our cost model. Therefore, we consider only the I/O cost for reading both of a page and a log record.

To assess the performance gain of our buffering scheme, we devise a mathematical cost model based on the I/O features shown in Table 1. As stated before, our performance gain mainly results from the amount of reduced updates, at the cost of writing and reading of log records. From the observations, we can drive a performance gain of P_{gain} as follows:

$$P_{gain} = N_d \times \{(C_u - C_{wl}) \times N_f - (C_{rl} - C_r) \times N_e\} \quad (1)$$
$$\approx N_d \times N_f \times (C_u - C_{wl} - C_{rl} - C_r) \quad (2)$$
$$\geq N_d \times N_f \times (C_{GC} - C_r) \quad (3)$$

Table 1. Considered parameters for our cost model.

Notations	Meanings or Definitions
C_u	I/O cost for updating a page
C_r	I/O cost for reading a page
C_{wl}	I/O cost for writing a log record
C_{rl}	I/O cost for reading a log record
N_d	Avg. # of dirty pages in the buffer pool
N_f	probability of virtual flushing of dirty pages
N_e	reload probability of pages with virtual flushing

In the cost model of (1), $\boldsymbol{N_e}$ and $\boldsymbol{N_f}$ is likely to be the same in a long span of time. Therefore, we can drive a simpler form of (2) rather than (1). In equation (2), the value of $\boldsymbol{C_u}$ varies with respect to various workloads of updates in the system and performance of used flash memory. Since the operation of an page update includes a write of page and a hidden cost for garbage collection, we get roughly $\boldsymbol{C_u = C_w + C_{GC}}$.

The value of $\boldsymbol{C_{wl}}$ is less than half the cost for a page write, i.e., $\boldsymbol{C_w}$, because we keep the size of a log record less than half the a page. In addition, $\boldsymbol{C_{rl}}$ is much less than $\boldsymbol{C_{wl}}$ in flash storage. Note that read speed is faster than the write speed in flash. Therefore, it is the case that $\boldsymbol{C_w \geq C_{wl} + C_{rl}}$. From this, we can reduce the cost model of (2) to that of (3).

In inequality (3), $\boldsymbol{C_{GC}}$ denotes the average cost for executing garbage collection that is caused by page updates. From many types of researches regarding to flash memory, such a garbage collection cost overhead is reported to be very expense because it leads to a number of page writes and block erases. Whereas, a page read is a very cheap operation in flash storage. Consequently, we can say that the proposed buffering scheme can efficiently significantly improve the I/O performance when it is used for handling frequent page updates with a large size of a buffer pool. This is because the performance gain is proportional with the number of dirty pages existing in the buffer pool as well as the frequency of virtual flushing.

5. Conclusion

In the paper, we proposed a flash-based buffering scheme that is efficient for reducing page updates via virtual flushing. Unlike the traditional buffering scheme, our proposed buffering scheme need not to actually flush any dirty page that is chosen as a victim at the buffer replacement time. Instead, we save a redo log record for that victim page before it is removed from the buffer pool without flushing. Additionally, when we need to write a long-living dirty page for the consideration of rapid recovery process, we save a redo log record without doing physical flushing. Through these two kinds of virtual flushing, our buffering scheme can replace an update operation with a small-sized pure-write. Since the cost of a pure-write in a log file is much less than that of a update operating, we can improve the I/O performance of database systems in the case of the usage of flash storage.

To evaluate the performance advantages of the proposed buffering scheme, we devised a theoretical cost model reflecting I/O features of flash storage. We developed the cost model based on the difference between the I/O cost for updating a page and the cost for virtual flushing. With the cost model-based evaluation, we showed that our scheme outperforms a traditional buffering scheme, when we take in mind flash-based

database systems. Such performance gains mainly come from hidden costs paid for out-of-place updates and garbage collection actions that are unavoidable in flash storage. Our proposed scheme seems to be a competitive flash-aware buffering scheme that is required for high-performance database systems.

Acknowledgement

This work was supported by the sabbatical program of the Dongduk Women's University in 2020.

References

[1] Stephan Baumann, Giel de Nijs, Michael Strobel, and Kai-Uwe Sattler, "Flashing Databases: Expectations and Limitations," In *Proc. of ACM Data Management on New Hardware*, pp.9-18, June 2010.
DOI: https://doi.org/10.1145/1869389.1869391

[2] Arul Selvan Ramasamy and Porkumaran Karantharaj, "RFFE: A Buffer Cache Management Algorithm for Flash-Memory-Based SSD to Improve Write Performance," Canadian Journal of Electrical and Computer Engineering, Vol. 38, No. 3, pp. 219–231, August 2015.
DOI: 10.1109/CJECE.2015.2431745

[3] Kai Zheng and Jian Wang, "Page Weight-Based Buffer Replacement Algorithm for Flash-Based Databases," In *Proc.* of International Conference on Computer Technology, Electronics and Communication, pp. 19-21, Dalian, China, December 2017.
DOI: 10.1109/ICCTEC.2017.00107

[4] Woon-Hak Kang, Sang-Won Lee, and Bongki Moon, "Flash-based Extended Cache for Higher Throughput and Faster Recovery," Journal of the VLDB Endowment, Vol. 5, No. 11, pp. 1615-1626, 2012.
DOI: https://doi.org/10.14778/2350229.2350274

[5] Gijun Oh and Sungyong Ahn, "Implementation of Memory Efficient Flash Translation Layer for Open-channel SSDs," International Journal of Advanced Smart Convergence, Vol.10, pp. 142-150, 2021.
DOI: http://dx.doi.org/10.7236/IJASC.2021.10.1.142

[6] Soyoon Lee and Hyokyung Bahn, "A Page Placement Scheme of Smartphone Memory with Hybrid Memory," The Journal of The Institute of Internet, Broadcasting and Communication, Vol. 20, No. 1, pp.149-153, 2020.
DOI: https://doi.org/10.7236/JIIBC.2020.20.1.149.

[7] Niv Dayan, Martin Kjær Svendsen, Matias Bjørling, Philippe Bonnet, and Luc Bouganim, "EagleTree: Exploring the Design Space of SSD-based Algorithms," Journal of the VLDB Endowment, Vol. 6, No. 12, pp. 1290-1293, 2013.
DOI: https://arxiv.org/abs/1401.6360

[8] Seong-Chae Lim, "A New Flash-based B+-tree with Very Cheap Update Operations on Leaf Node," In *Proc.* of ETBDA, Bangkok, Thailand, 2016.
DOI: https://doi.org/10.5392/JKCA.2016.16.08.706

[9] M. Ganim, G. A. Mihaila, B. Bhattacharjee, K. A. Ross, and C. A. Lang, "SSD Bufferpool Extensions for Database Systems," In *Proc.* of VLDB, Vol. 3, No. 1-2, pp. 1435-1446, September 2010.
DOI: https://doi.org/10.14778/1920841.1921017

[10] Chu Li, Dan Feng, Yu Hua, and Wen Xia, "Gasa: A New Page Replacement Algorithm for NAND Flash Memory," In *Proc.* of IEEE International Conference on Networking, Architecture and Storage, pp. 1-9, 2016.
DOI: 10.1109/NAS.2016.7549403

[11] Kyosung Jeong, Sungchae Lim, Kichun Lee, and Sang-Wook Kim, "A Flash-Aware Buffering Scheme with the On-the-Fly Redo for Efficient Data Management in Flash Storage," Computer Science and Information Systems, Vol. 14, No. 2, pp. 369-392, 2017.
DOI: 10.2298/CSIS160830014J

[12] Sungup Moon, Sang-Phil Lim, Dong-Joo Park, and Sang-Won Lee, "Crash Recovery in FAST FTL," In *Proc.* of Software Technologies for Embedded and Ubiquitous Systems, pp. 13-22, October 2011.

DOI: https://dl.acm.org/doi/10.5555/1927882.1927888
[13] Fang WangGuanying Wu and Xubin He, "Delta-FTL: Improving SSD Lifetime via Exploiting Content Locality," In *Proc*. of ACM European conference on Computer Systems, pp. 253-266, Bern, Switzerland, April 2012. DOI: https://doi.org/10.1145/2168836.2168862
[14] Yinan Li, Bingsheng He, Robin J. Yang, Qiong Luo, and Ke Yi, "Tree Indexing on Solid State Drives," Journal of the VLDB Endowment, Vol. 3, No. 1-2, pp.1195-1206, September 2010. DOI: https://doi.org/10.14778/1920841.1920990
[15] Gap-Joo Na, Sang-Won Lee, and Bongki Moon, "Dynamic In-Page Logging for B+-tree Index," IEEE Transactions on Knowledge and Data Engineering, Vol. 24, No. 7, pp.1231-1243, July 2012. DOI: 10.1109/TKDE.2011.32
[16] C. Mohan and F. Levine, "ARIES/IM: An Efficient and High Concurrency Index Management Method using Write-ahead Logging," In *Proc.* of SIGMOD, pp. 371-380, San Diego, USA, 1992. DOI: https://doi.org/10.1145/130283.130338
[17] J. Do, D. Zhang, J. M. Patel, D. J. DeWitt, J. F. Naughton, and A. Halverson, "Turbo-charging DBMS Buffer Pool using SSDs," In *Proc*. of ACM SIGMOD, pp. 1113-1124, Greece, June 2011. DOI: https://doi.org/10.1145/1989323.1989442
[18] Sungjin Lee, Dongkun Shin, and Jihong Kim, "BAGC: Buffer-Aware Garbage Collection for Flash-Based Storage Systems," IEEE Transactions on Computers, Vol. 62, No. 11, pp. 2141-2154, 2013. DOI: 10.1109/TC.2012.227
[19] URL Link: *https://www.digitalcitizen.life/simple-questions-what-trim-ssds-why-it-useful*, "What is SSD TRIM, why is it useful, and how to check whether it is turned on," July 2019.
[20] Laura M. Grupp, John D. Davis, and Steven Swanson, "The Bleak Future of NAND Flash Memory," In *Proc*. of the 10th USENIX Conference on File and Storage, San Diego, USA, 2012. DOI: https://dl.acm.org/doi/10.5555/2208461.2208463