Computer Science and Information Systems 14(2):369–392 DOI: 10.2298/CSIS160830014J

A Flash-Aware Buffering Scheme with the On-the-Fly Redo for Efficient Data Management in Flash Storage

Kyosung Jeong[1], Sungchae Lim[2], Kichun Lee[3], and Sang-Wook Kim[4]

[1] Dept. of Computer and Software, Hanyang University, Korea
ksjeong@saltlux.com
[2] Dept. of Computer Science, Dongduk Women's University, Korea
sclim@dongduk.ac.kr
[3] Dept. of Industrial Engineering, Hanyang University, Korea
skylee@hanyang.ac.kr
[4] Dept. of Computer and Software, Hanyang University, Korea
wook@hanyang.ac.kr

Abstract. Thanks to remarkably fast random reads and rapidly decreasing prices per bit, flash storage has been regarded as a promising alternative to traditional hard disk drives (HDDs). Although flash storage has many distinguished hardware features, it still suffers from the poor I/O performance in the case of update operations. Due to the absence of in-place updates, differently from HDDs, flash storage needs to modify data through out-of-place updates. For this reason, it is required to continuously renew the mapping information between a logical page address and its new physical address, invalidating its old physical address. When the invalidated pages swallow most of free space in flash storage, the actions of garbage reclamation are needed. Since the actions of garbage reclamation are very costly, it is crucial to reduce the number of update operations for the use of flash storage in enterprise-scale database systems. In this light, we propose a new buffering scheme that evicts dirty pages without writing them to storage, thereby reducing the amount of update operations considerably. That is, our buffering scheme enables the flushing-less evictions of dirty pages. To correctly read a page undergoing its flushing-less eviction, we propose a new on-the-fly redo mechanism that enables restoring the lost updates of the page in normal database processing. For fast execution of the on-the-fly redo, we maintain memory-resident log data of a reasonable size. To show the performance advantages of the proposed scheme, we performed extensive experiments based on the TPC-C benchmark, by running them on the open-sourced Berkeley DB equipped with/without our scheme. The results show that our scheme yields a much better performance by reducing the amount of page updates significantly.

Keywords: flash storage, buffering scheme, database system, recovery

1. Introduction

During the past decade, flash storage has rapidly widened its application domains as storage for diverse portable devices such as laptops, digital cameras, and smart phones. The popularities of flash storage are largely due to its salient features of fast random

reads, low power consumption, and good shock resistance [1][2]. Although the cost per bit of flash storage is still higher than that of traditional hard disk drives (HDDs), technology enhancements in the future seem to narrow the per-bit cost gap rapidly [7][9] [15]. In this light, much research has been done to incorporate flash storage into enterprise-scale systems for the purpose of performance improvement. Most of such research was devoted to solving the flash storage's problem of severe asymmetric performances of read operations and update operations [6][7][8][9][10][11][15][16].

Unlike the HDDs, flash storage has the hardware constraint of *erase-before-write* due to its disability of in-place updates, which is the major cause of expensive update operations in flash storage. Because writing of data is allowed only on an empty page, flash storage has latent I/O overheads for performing out-of-place writes and block merging needed for garbage reclamation [1][12][19]. To minimize performance degradations from such I/O overheads in flash storage, researchers have proposed a number of efficient algorithms for the flash translation layer (FTL) [19][20]. Owing to the FTL, flash storage can provide the HDD-like I/O interfaces by managing the mapping information between the logical addresses and physical addresses of pages. Although the use of the FTL may diminish the occurrences of *full* block merges during garbage reclamation, it cannot guarantee a robust I/O performance in the presence of a large volume of random writes (i.e., random updates) [4][6][8][15]. To avoid the limitation of such FTL-based solutions, this paper addresses a new buffering scheme that can reduce the number of update operations significantly (footnote: The preliminary version of this paper has been presented as a short paper (4 pages) in ACM CIKM 2015 [15].)

There have been many research efforts on effective buffering for HDD-based DBMSs [4][7][23]. In the case of HDDs, the I/O time for wring a page is nearly the same as that for reading a page, and data can be updated in-place. Therefore, there is no reason to make an intense effort for reducing page updates at the expense of increased page reads in HDD storage. As a result, the key design goal of buffering schemes with HDDs is to enhance the buffer hit rate for the purpose of a less number of page reads [23]. In this context, the LRU (least recently used) algorithm and its variations are widely accepted as buffer replacement algorithms [4][17].

Under the LRU algorithm, if a dirty page X is picked as a victim page to get a free frame from the buffer pool, the current in-buffer image of X is written to storage before its eviction. In traditional buffering schemes devised for HDDs, such an eviction with data flushing is common and reasonable from the performance point of view [13][23]. However, in the case of flash storage, it may not be the case due to its cheap cost of page reads. For instance, assume that we evict the dirty page X without flushing it. Since the read cost for reloading X is very cheap in flash storage, the cost for restoring X may be also cheap. This makes the *flushing-less eviction* of X lucrative in flash storage by eliminating an update on X, at the expense of a page read for reloading X. Based on this idea, our buffering scheme enables the flushing-less evictions of dirty pages in the buffer pool.

Since such flushing-less evictions make a database inconsistent, we need a mechanism for correctly restoring a dirty page that underwent a flushing-less eviction. For this purpose, we rely on the redo mechanism originally devised for the recovery purpose. For exposition, consider that dirty page X is evicted without flushing. When X is about to be read again, our buffering scheme reads the in-storage (old) image of X and then restores the correct image of X by applying X's redo log record(s) on that. We refer

to such a real-time redo action on *X* as the *on-the-fly* redo. For fast executions of the on-the-fly redo, we maintain a portion of redo log records in main memory. Although the on-the-fly redo entails more page reads and consumes some more CPU times, such overhead can be compensated by the I/O benefits from the decrease of page updates. Besides the reduction of page updates, our proposed buffering scheme provides two additional advantages. First, it can be used for enabling the no-steal policy in buffer management [12][13][23]. If any traditional buffering scheme attempts to support the no-steal policy, it may suffer from restrictive buffer replacements because the dirty pages affected by *uncommitted updates* cannot be evicted from the buffer pool. Here, the uncommitted update means an update made by a transaction that has not committed yet. This restrictive selections of victim pages are apt to cause exhaustions of free buffer space or frequent buffer misses [13][23]. Unlike that, our buffering scheme can select a dirty page as a victim while providing the no-steal policy, because that dirty page can be simply evicted without flushing. Second, the mechanism of the on-the-fly redo can be used to shorten the time for reincarnating the buffer pool after system failure. To restart the buffer pool, the recovery algorithm is usually required to perform the redo phase to restore the up-to-date images of dirty pages that were buffered at the time of system failure [13]. Redo actions are executed using the redo log data. In the case of our recovery algorithm, however, it can restart the buffer pool in a faster way by reading a set of log records without performing actual redo actions. Then, the actual redo actions are gradually performed through the on-the-fly redo during normal database processing. As a result, our buffering scheme has an advantage of faster recovery.

The rest of this paper is organized as follows. In Section 2, we address technical challenges related to our work, and explain the research motivations of this paper. In Section 3, we propose our buffering scheme that supports flushing-less evictions. Then, we present the recovery algorithm for the proposed buffering scheme in Section 4. We verify our performance advantages by performing experiments with the TPC-C benchmarks in Section 5. Finally, we summarize and conclude the paper in Section 6.

2. Background

2.1. Traditional Buffering Schemes

Because of the poor speeds of random reads in HDDs, the use of a memory buffer pool is indispensable for enhancing the I/O performance of disk-based DBMSs [15][16][23]. To improve the buffer hit rate of the buffer pool with a limited size of main memory, the algorithms for efficient buffer replacement were intensively studied [7][16][23]. Among them, the LRU algorithm has been widely used due to its cheap run-time cost and competitive hit rate. If the LRU algorithm and its variants pick a dirty page as a victim at the time of buffer replacement, it needs to flush that page to storage before the page's eviction. Since the updated page may contain uncommitted updates, the write-ahead-logging (WAL) protocol is usually employed [12][13][23]. Most modern DBMSs adopt the WAL protocol and the LRU buffer replacement for achieving high I/O performance.

Because our proposed buffering scheme is designed based on the redo mechanisms used for the system recovery, some knowledge of the recovery algorithm is necessary

for understanding our key idea. For this reason, we briefly describe the ARIES algorithm [13]. ARIES snapshots a consistent database state by storing a *dirty page table* and the list of in-progress transactions into the log file. When the recovery procedure is initiated to handle system failure, ARIES accesses the last checkpoint record. Then, the three phases of a log analysis, redo actions, and undo actions are consecutively performed [13][15][23]. During the log analysis phase, ARIES identifies the set of *loser* transactions to be rolled back. Then, the redo phase begins to repeat the lost updates in databases using the redo log data. Lastly, ARIES performs the undo phase to revoke the uncommitted updates made by the loser transactions. For that, AIRIES uses the log-sequence-number (LSN) that is stamped on every data page and log record. In general, an LSN is the same as the log file offset where the log record with that LSN is stored. By comparing the LSNs of a log record *R* and its involved data page, ARIES determines whether or not the update operation recorded in *R* has already been reflected on that page [13].

When the ARIES algorithm is used for disk-based DBMS, the redo phase accounts for the most portion of the recovery time because this phase requires a large number of random reads while restoring dirty pages in the buffer pool [13][17][21]. For this reason, the ARIES algorithm usually flushes aged dirty pages during normal database processing [13]. Because the update operation is not a special concern in HDD storage, the mechanism of the dirty page flushing seems to be lucrative for the disk-based DBMS, if we take into account the reduced redo phase time. However, this cannot be the case for the flash-resident DBMS because of its high costs for page updates. This motivates us to devise a novel buffering scheme that supports the fast recovery without frequent flushing of dirty pages.

2.2. Earlier Flash-based DBMSs

Since data accesses in flash storage are performed without the mechanical movement of arms occurring in the HDD, the flash-based solid state device (SSD), which is composed of multiple flash dices and an internal H/W controller, provides the advantages of fast and uniform speeds of random reads [2][6][8]. The read speed of flash storage is above an order of magnitude faster than that of HDD storage. Writing of data into an empty page of flash storage has 9~10 times faster speeds than cutting-edge HDD storage. In order to utilize flash storage more effectively for achieving higher I/O performance in a DBMS, a number of research efforts have been made in the database community [4][5][6][7][8][9][10][11][12][15][16][17][18][19][20][21][22].

As one of such efforts, there exists research that accommodates flash storage as a cache area located between a memory buffer pool and HDD storage [4][5]. In this research, the concept called the region temperature is invented for the purpose of estimating the degree of the benefit from caching a data region in flash storage. To this end, the authors of [4] logically divide the database area in HDD storage into equal-size regions and compute their *temperatures*. The temperature of region *R* is computed by accumulating the I/O benefits obtainable by caching that region in flash storage. Then, the region temperatures are used for determining caching priorities of regions.

Because data are duplicated across flash storage and primary HDD storage, it is required to pay some extra costs for preserving data consistency between them. In particular, it is important to retain database consistency in the face of system failure. As

a naïve mechanism for recovery, one may invalidate the whole data cached in flash storage at the time of system restart [4]. With this approach, however, the recovered system usually suffers from severe performance degradations until the invalidated hot regions are re-cached in flash storage. To shorten such a ramping-up interval, some ideas have been proposed [5]. However, as the flash-cache approaches [4][5] read a majority of data from HDD storage, they do not seem to enhance the DBMS performance fundamentally.

To solve the problem in flash-cache approaches, some research attempts to accommodate flash storage as a primary storage area, rather than a cache area. For this, the journaling mechanism [14][23] is adopted to update a data page at a cheaper I/O cost. For instance, IPL proposed in [11] stores databases on flash storage and prepares a log area at the tail of each flash block. If an update arises in any data page within a block *B*, IPL writes the corresponding log data into the log area of *B*. That is, IPL performs out-of-place updates by itself without the help of the FTL. When the log area becomes full because of a number of updates on its associated data pages, a block merge operation is conducted to clean that used log area again. That is, a block erase operation is executed, and all the data pages in the block are rewritten with their up-to-date images. Because the block merging can be done within a single block, its price is very low. For this reason, the IPL algorithm ensures good I/O performance for the flash-based DBMS with a number of update operations. Since IPL does not support the mechanisms of the system recovery and the concurrency control for transactions, however, the transactional IPL (TIPL) was proposed to extend the IPL algorithm for the use in transaction processing systems [12].

Although IPL and TIPL can avoid severe performance degradations caused by frequent update operations in flash storage, they can easily suffer from considerable I/O overheads for scanning the log areas. To correctly read a data page *P* from a block *B*, these methods need to scan the whole log area in *B* so as to check the existence of any log data associated with *P*. If the log data are found, they are applied to generate the up-to-date image of *P*. That is, some redo action is required for reading page *P* correctly. Since the log scanning is always executed for loading a new page up to the buffer pool, it can harm the I/O performance of the flash-based DBMS when buffer misses occur frequently.

Besides such costs for redo actions, the IPL and TIPL algorithms have the problem of the wastes of flash storage space because of the existence of log areas in blocks. To reduce the storage wastes for logging, they rely on a special I/O interface that can support a finer-granularity of log writes. That is, it is assumed that 0.5KB worth of a sector is used as the unit of log writing, rather than the usual unit of 2~4 KB for data writing. However, such a smaller unit of writing is not supported in MLC (Multi-Level Cell) flash, which is the most popular type in the market for their low prices per bit [6][15]. Due to such H/W restrictions and overheads for redo actions, IPL and TIPL seem to be unsuitable for flash-resident databases. To overcome such problems, our research focuses on the update-aversive buffering scheme that can run well on off-the-shelf flash storage devices.

3. Proposed Buffering Scheme

In this section, we first present our proposed mechanism for the on-the-fly redo, which is used to enable flushing-less evictions of dirty pages in the buffer pool. Then, we address how to efficiently handle the commits and aborts of transactions by using the on-the-fly redo.

3.1. On-the-fly Redo

As for the buffering scheme for traditional disk-based DBMSs, its usage is mainly aimed at reducing the number of disk reads by caching hot/warm data pages in main memory [7][23]. For this, the traditional buffering scheme usually adopts the LRU-style algorithms as its buffer replacement policies. If a dirty page is chosen according to an LRU-style algorithm for buffer replacement, the current in-buffer image of that page is written to storage before its eviction. Through the flushing of the dirty page, the database consistency is guaranteed in ease. Since the update cost is not tremendous in HDD storage, it seems to be reasonable to flush dirty pages at their eviction times. However, it cannot be true in the case of flash-resident databases because update costs are too expensive in flash storage [7][8][9][15].

In this context, we propose a flash-aware buffering scheme that provides the ability of flushing-less evictions of dirty pages, thereby diminishing the occurrences of page updates. To restore the correct image of a dirty page that has been evicted without flushing, correspondingly, we incorporate a real-time recovery mechanism into the proposed buffering scheme. Specifically, we perform a redo action on that page having a flushing-less eviction in such a way that the lost updates on it due to flushing-less eviction is recovered in online mode. Because the redo mechanism is originally devised to preserve the database consistency in face of system failure, the log data used for it are stored in storage and are retrieved in off-line mode. In the proposed scheme, however, we manage a portion of redo log data in main memory and index them for their fast accesses. In this paper, we refer to this online mode redo, performed during normal database processing, as the *on-the-fly* redo.

To explain how to execute the on-the-fly redo, we use Figure 1. The buffer pool consists of a set of buffer frames, a page/frame addressing table (PFAT), and an in-memory hash table used to manage log data. Here, we refer to this in-memory hash table as the *online log table*. The PFAT contains the mapping information between the logical addresses of buffered pages and their reserved frame IDs. The log records in the online log table are indexed with respect to the logical addresses of their associated dirty pages. Because the proposed method does not require undo actions for aborted transactions, we do not save undo log data either in storage or in the online log table. Therefore, we refer to the log records of the online log table the redo log records.

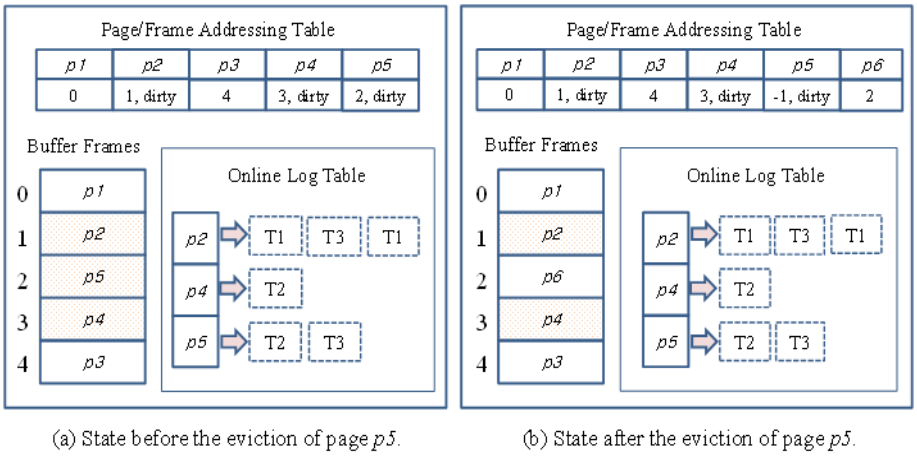


Figure 1. The way for managing the proposed buffer pool. The dirty page *p5* is evicted without flushing.

Figure 1(a) shows the buffer pool state before page *p5* is evicted from the buffer pool. Here, we assume that the buffer pool can cache up to five pages and it is currently full of buffered pages. Among the five pages, the pages of *p2*, *p4*, and *p5* are marked as dirty pages in the PFAT. In the case of *p5*, it is a dirty page using the buffer frame of number 2 and has been updated by transactions *T2* and *T3* in this order. To address the way to evict page *p5* without flushing, we assume that page *p6* is requested at the time of Figure 1(a). Since there is no free buffer frame now, the buffer replacement algorithm is executed to pick a victim page to be evicted. Suppose that page *p5* happens to be selected as the victim.

In the case of a usual buffering scheme, the current in-buffer image of *p5* is written to storage for database consistency. Unlike that, to avoid a page update, our buffering scheme just evicts *p5* without its flushing. Instead, we change the PFAT entry of *p5* as in Figure (b), where *p5*'s PFAT entry is modified with the value of (-1, dirty). Note that the minus value in the frame number expresses that there exists a flushing-less eviction on *p5*. Then, the newly buffered page of *p6* replaces the buffer frame of *p5*.

To explain how to correctly restore the page with flushing-less eviction, suppose that transaction *T1* requests a record in *p5*. To process the I/O request of *T1*, our buffering scheme first looks into the PFAT entries so as to check the existence of *p5* in the buffer pool. From the PFAT entry of *p5* in Figure (b), it is found that *p5* experienced flushing-less eviction. Thus, the on-the-fly redo is applied to *p5*. That is, our buffering scheme reads *p5* from storage to the buffer and applies its two redo log records to *p5*. Thanks to the page-granularity on-the-fly redo, we can restore the correct image of *p5* for *T1*. To make the restored image of *p5* valid, our buffering scheme changes the minus frame number of *p5* with its new frame ID.

The correctness of the on-the-fly redo mechanism can be shown as follows. Since our redo log data is made according to the *physiological logging* policy, it records an update operation occurring *only within* a single page. Here, the update operation is expressed with a logical operation [12][13]. Based on the property of the physiological redo log, it is the case that the recent image of any page *P* can be correctly generated by chronologically applying the associated physiological log records on *P*. Moreover, with

a proper locking protocol, any of permutations of multiple physiological log records can restore the correct image of an updated page. Therefore, our on-the-fly redo mechanism is correct and thus can be safely applied.

If we attempt to support the no-steal policy in the same way as in the traditional buffering scheme, we cannot evict any dirty pages having uncommitted updates at the time of buffer replacement. Such restriction on victim selections usually entails poor buffer hit rates. In particular, the existence of some long living transactions may severely hurt the buffer hit rate by exhausting free buffer space when they update a number of pages during their long life [13][23][15]. This is the reason why the steal policy is common in the traditional disk-based DBMSs. Since our buffering scheme enables to evict dirty pages without flushing, it can easily support the *no-steal policy* in buffer management. Due to the advantages of such a no-steal policy, the proposed buffering scheme not only provides the faster recovery at the time of system failure but also eliminates the overhead of maintaining undo log records during normal database processing. More details will be elaborated in Section 4.

3.2. Handling of Transaction Commits

In the conventional disk-based DBMSs, the buffered log writing is useful for reducing the I/O overheads required in storing log data. That is, new log records are appended to an in-memory log buffer, which is written into the log file through a single I/O call when it gets full of log records. To commit a transaction *T*, therefore, the log buffer should be flushed out to write all the log records associated with *T* [3][17][23]. Note that all the committed log records should be stored in safe storage for executing redo actions for the recovery. Since the log buffer usually contains the log records owned by other transactions, uncommitted log records could be flushed along with the committed log records of *T*. Here, an uncommitted log record indicates a log record related to an uncommitted update. Such uncommitted log records are those for undo actions and thus are called the *undo log records*. They are normally used for rolling back the update operations made by aborted transactions.

Since the no-steal policy can be effectively realized with our scheme, there is no need of undo log records. For this reason, when a transaction *T* requests its commit, we save only the log records owned by *T* without saving other transactions' log records. Specifically, we copy the log records belonging to *T* from the online log table into the log buffer. Then, the log buffer is flushed in order to store *T*'s committed log records. Since the total amount of log records for a single committed transaction is normally much less than the smallest size of data writes in flash storage, we may waste a considerable fraction of I/O bandwidth while recording log data.

To alleviate the I/O wastes in logging, we adopt the well-known technique called the *group-commit* [23] as an optional commit protocol, together with the ordinary *immediate commit* protocol. Using the group-commit protocol, we can gather the committed log data in the log buffer by delaying times of transactions' commits. When the log buffer becomes full of log records, its data is flushed out to storage, by following the group-commit protocol. To prevent excessive delays of transactions' commits, we set the maximum delay time for our group-commit protocol.

Although the log records of a transaction *T are stored at its commit time*, we do not remove their duplicates managed in the online log table. This is because they are still

required for the correct executions of the on-the-fly redo. For the exposition, suppose that transaction *T2* of Figure 1(b) has committed and its log records are all written to the log file. If we remove the committed log records of *T2* made for page *p5*, the on-the-fly redo cannot be correctly performed on *p5* because of the absence of the committed log records of *T2*. Therefore, the committed log records of *T2* need to be kept for *p5*. For that reason, our buffering scheme retains committed log records in the online log table until the on-the-fly redo can be correctly performed without them. More specifically, the deletion of committed online log records is performed at the time of checkpointing. In the proposed scheme, the checkpointing process deletes committed log records, and also flushes their related dirty pages. The algorithm for the checkpointing process is presented in Section 4.

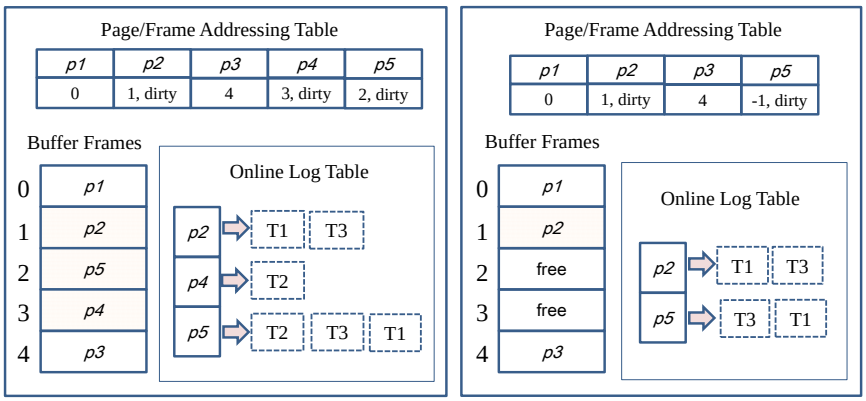
3.3. Handling of Transaction Aborts

Thanks to the no-steal policy implemented in our buffering scheme, our recovery process has no need of undo actions after system. However, during normal database processing, we need to handle individual transaction aborts correctly to nullify their updates made on the buffered pages. Let us assume that a transaction *T*, which has updated some database pages, is about to abort itself. In the buffering scheme used in disk-based DMBSs, the uncommitted updates of *T* are nullified with undo actions on their involved dirty pages in the buffer. Thus, for logging these undo actions, the compensation log records (CLRs) are saved in the log file during the *T*'s abort [13]. The use of CLRs inevitably enlarges the log data and consumes more CPU times for applying the undo log records to dirty pages.

Unlike that, our proposed buffering scheme eliminates the necessity of the CLR usage. To explain this, we use Figure 2, which illustrates how the states of the PFAT and the online log table are modified for aborting a transaction. Figure 2(a) depicts a situation where transactions *T1*, *T2*, and *T3* are in progress and they have made updates on pages *p2*, *p4*, and *p5*. According to the no-steal policy for buffer management, those updates have not been reflected on storage until now. Suppose that transaction *T2* aborts itself at the time of Figure 2(a). In the case of the traditional buffering scheme, undo actions are performed to nullify *T2*'s update operations on *p4* and *p5*, along with the saving of the corresponding CLRs. Unlike this conventional mechanism, we just evict the two pages *p4* and *p5* updated by *T2* and delete the online log records owned by *T2* as in Figure 2(b). During the abortion of *T2*, we neither write any CLRs nor perform undo actions on *p4* or *p5*. We just free the buffer frames reserved for the updated pages without any actual updates

The correctness of the above mechanism for the transaction abort can be easily verified. Suppose that a certain transaction asks for page *p4* after the transaction abort of Figure (b). Since the PFAT does not contain any entry for *p4*, that page will be read from storage. Since the no-steal policy is employed in our scheme, it is ensured that the in-storage image of *p4* was not affected by *T2*. Therefore, the transaction can correctly read page *p4*. In turn, suppose that page *p5* is requested by a certain transaction. For this data request, our buffering scheme executes the on-the-fly redo on *p5* because the PFAT entry of *p5* contains the frame number of a minus value. Since the online log records of *T2* were deleted already, page *p5* can be correctly restored by using the online log

records of *T3* and *T1*. Consequently, we can clean the update operations made by transaction *T2* without the use of undo actions and logging of CLRs.



(a) State before transaction *T2* aborts. (b) State after transaction *T2* has aborted.

Figure 2. The way to abort transaction *T2*. For the abortion of *T2*, pages *p4* and *p5* are evicted.

Due to the mechanism of the on-the-fly redo, the proposed buffering scheme can efficiently process transaction aborts without any page updates or log writing, which are required in the conventional buffering schemes. However, our scheme needs extra page reads used for re-caching the pages that have been evicted for transaction's abortion. For example, in the case of *p5* in Figure 2, we need to read this page before an on-the-fly redo is applied to it. As mentioned before, the I/O cost for reading a page is very low in the case of flash storage. As a result, the transaction abort can be handled at a cheaper price by our buffering scheme, compared with the traditional scheme. The performance advantage is discussed in Section 5.

4. Algorithms for Recovery

In this section, we present how to recover the buffer pool after system failure. To expedite the system recovery, we rely on the checkpointing scheme that is usually employed to create a snapshot of the recent database state. However, the proposed checkpointing algorithm performs the compaction of the online log table as well as the creation of a recent database snapshot.

4.1. Checkpointing Scheme

4.1.1 Overview

Checkpointing in a DBMS aims at creating a snapshot of a continuously changing state of a database that is used for faster recovery after system failure [13]. Among the earlier

checkpointing schemes, the fuzzy checkpointing scheme has been widely accepted for its low I/O overhead and short duration. To make a database snapshot, it records the logical addresses of the dirty pages residing in the buffer pool and the LSNs of their corresponding *recovery log records*. Here, the recovery log record of page *P* is defined as the log record created for the first update made on *P* after *P* has been buffered. The dirty page list and the LSNs of the recovery log records are stored together in the *dirty page table*. Besides the dirty page table, the fuzzy checkpointing algorithm saves the IDs of in-progress (live) transactions in the checkpoint record.

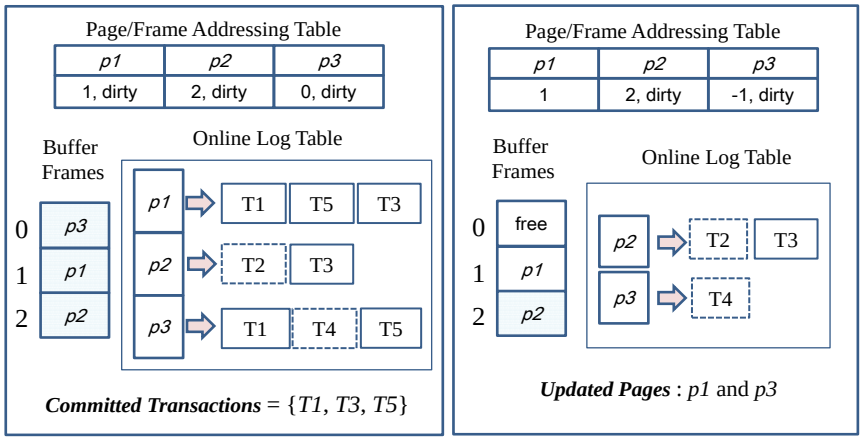
To explain the usage of the LSN information of recovery log records, let us assume that a page *P* has been updated three times within the buffer pool. Let the update times of the dirty page *P* be *t1, t2*, and *t3*, respectively, after *P* has been buffered. According to the WAL protocol, the same number of log records are created for the recovery purpose. In this case, the first log record representing the update at time *t1* is used as the recovery log record of *P*. *P*'s log records preceding the recovery log record have been already reflected in *P* and thus are useless for redo actions on *P*. For this reason, the recovery log record becomes the starting point of redo actions for *P* in the presence of system failure. Therefore, the minimum (i.e., oldest) LSN of the recovery log records for all the updated pages is used as the starting point of the redo phase. That is, the redo phase is conducted by scanning all the log records from this minimum LSN to the end of a log file. Recall that the LSN of a log record is usually represented as the log file offset where the log record is stored. In this paper, the minimum LSN is called the *redo starting point*.

To shorten the checkpointing time, the fuzzy checkpointing scheme does not enforce flushing of all the dirty pages. Therefore, the redo starting point may stay backwards far from the present if there exist too aged (long-living) dirty pages in the buffer. To prevent the redo phase from being prolonged, the traditional buffering schemes based on the fuzzy checkpointing usually take an approach that flushes gradually dirty pages during normal database processing, rather than during checkpointing. That is, a background process is periodically invoked to flush long-living dirty pages for fast recovery [23]. This helps the traditional buffering schemes to reduce its recovery time at the cost of an increasing number of page updates during normal database processing.

Unlike the previous fuzzy checkpointing schemes, our checkpointing scheme has another mission of compacting the online log table, as well as creating a database snapshot. To see this, we use Figure 3(a) where there exist three dirty pages in the buffer pool. Currently, pages *p1, p2*, and *p3* have three, one, and two committed log records, respectively, in the online log table. Since the copies of those committed log records have been already stored in the log file, we can delete them. When deleting those committed log records, however, we have to carefully update their involved dirty pages. As stated earlier, the careless deletions of committed online log records entail incorrect executions of on-the-fly redo actions.

When the proposed checkpointing scheme is applied to the buffer state in Figure 3(a), five committed log records are eliminated as in Figure 3(b). In this example, we assume that the log compaction is allowed only for the dirty page having two or more committed log records involved with it. In the case of *p1* in Figure 3(a), its three log records are all committed ones. Therefore, the current in-buffer image of *p1* can be written to storage without impairing the no-steal policy, and its online log records can be safely deleted. For this, the PFAT entry of *p1* is updated as in Figure 3(b), where the state of page *p1* is changed to a clean page. On the other hand, page *p2* has a single

committed log record in the online log table of Figure 3(a). Thus, we do not perform log compaction on *p2*. As a result, the buffer state of *p2* remains intact as in Figure 3(b). Through the restrictive selections of the dirty pages to be flushed, we can reduce the total number of page updates while executing the actions for online log compaction.



(a) Buffer state before checkpointing. (b) Buffer state after checkpointing.

Figure 3. Log compaction at the checkpointing time. Five committed log records are deleted and two pages are updated for the online log compaction.

In the case of page *p3*, a sophisticated mechanism is required for log compaction. As shown in Figure 3(a), page *p3* has an uncommitted online log record, together with two committed ones. Because our checkpointing scheme conforms to the no-steal policy, it cannot write the in-buffer image of *p3* to storage now. Rather, we read the in-storage image of *p3* and perform an on-the-fly redo on *p3* by using the two committed log records. Then, the after-redo image of *p3* is written to update *p3*. Correspondingly, *p3*'s PFAT entry is modified so that the on-the-fly redo is performed on *p3* using the remaining online log record. Recall that the similar mechanism was employed for handling transaction aborts in Section 3.3.

After the actions for log compaction, the checkpointing algorithm needs to create a dirty page table. In the case of Figure 3(b), page *p2* is saved in the dirty page table. As the recovery record of *p2*, we record the first committed log record that is owned by transaction *T3*. On the other hand, page *p3* is not included in the dirty page table. Because the proposed buffering scheme employs the no-steal policy, the update operations in the uncommitted log records never affect the database state. Therefore, we can regard the dirty page such as *p3* to be a clean page from the consideration of system recovery. Then, the in-progress (i.e., live) transaction list is saved in the checkpoint record, along with the dirty page table.

4.1.2 Checkpointing Algorithm

The major purpose of the traditional checkpointing algorithms is to make a snapshot for the fast recovery after system failure. Through periodic checkpointing, the previous checkpointing algorithms prevent the recovery time from enlarging too much. Being slightly different from them, our checkpointing algorithm does not take such a periodic approach. This is because we can keep the recovery time very short owing to the capability of gradual recovery actions that can be performed along with normal transaction processing. Rather, the execution of our checkpointing algorithm depends on the amount of free space in the on-line log table. More specifically, in the range of 5% to 10% of free space in that table, we begin the checkpointing procedure for log compaction as well as for flushing dirty pages to storage.

At the time of checkpointing, we first create a log record denoting the start of checkpointing. Next, the actions for log compaction are performed to clean the committed log records from the online log table. During the log compaction, the checkpointing algorithm can flush some dirty pages so that the associated committed updates are reflected on the database. Subsequently, the information of the dirty page table and in-progress transactions is saved in a new checkpoint record. Finally, the master record in the system is modified to correctly point to the newly created checkpoint record. These stepwise actions are not different from those in traditional disk-based DBMSs, except for the extra actions for log compaction.

Figure 4 gives the pseudo code for the proposed checkpointing algorithm. The algorithm takes the two input parameters of *min_del* and *max_age*. The former parameter says the minimum number of committed online log records where log compaction can be executed with respect to a single page. That is, by setting *min_del* to *k*, log compaction for a dirty page *p* is delayed until *p* has *k* committed online log records. Because at least *k* updates per page are required to incur a single update operation, we can reduce the total amount of page updates in flash storage. With a greater *min_del*, a larger number of updates could be reflected into storage via a single I/O. In fact, we pick 16 as the value of *min_del*. In the case of a cold page, however, the number of updates on it could not be large enough for log compaction. In an extreme case, a set of dirty pages with a small number of committed online log records may swallow the memory available for the online log table. Against that, the parameter of *max_age* is introduced. By deleting the committed online log records that are older than *max_age*, our algorithm could perform the log compaction even for a dirty page having a very low update frequency. By adjusting these two parameters appropriately, we can control the update frequency, while preventing the continuous increase of the online log table size.

The checkpointing algorithm of Figure 4 conducts its initial steps in lines 1-2. After the initial steps, it performs the actions of log compaction in lines 4-19, with respect to each dirty page existing in the buffer pool. Using the hashed information of the online log table, the algorithm counts the number of committed online log records of page *P* in line 5. If this number is not zero, the algorithm decides whether log compaction is necessary or not in line 8. If log compaction on *P* is not lucrative, then page *P* is recorded in the dirty page table without log compaction. That is, the logical address of *P* and the LSN of its recovery record are recorded in line 19. Otherwise, the actions for log compaction are performed in lines 9-17.

Algorithm 1: Procedure *CreateCheckpointRecord*

input: *min_del* = minimum *No.* of committed log records to be deleted;
max_age = restriction on the flushing-less buffering time of dirty pages;

```
1  SaveLogRec(LogFile, "Beginning of a Checkpoint");
2  Create a new dirty page table for checkpointing and let it be denoted by DT;
3  foreach page P ∈ Online_Log_Table do // actions for each dirty page
4      L ← GetListLogReccords(P); // L is used to point to the list of the
       online log records (LRs) of P
5      N_c ← GetNumCommitRec(L); // get the number of committed LRs of P
6      if N_c ≥ 1 then
7          I_f = GetIdxFirstCommitRec(L); // index of the oldest committed LR in L
8          if N_c ≥ min_del or (CurrentLSN − L[I_f].LSN) ≥ max_age then
9              if N_c = |L| then // P has no uncommitted LRs
10                 UpdatePage(P, Frame(P)); // update P with its in-buffer image
11                 Set the dirty bit of P's PFAT entry to zero;
12             else
13                 OldImage ← ReadPage(P); // page read from flash storage
14                 Image ← RedoWithCommit(OldImage, L); // use of on-the-fly redo
15                 UpdatePage(P, Image); // occurrence of a page update on storage
16                 Free the buffer frame of P and replace P's PFAT entry with (−1, dirty);
17             DeleteCommitRec(Online_Log_Table, P, N_c); // log compaction for P
18         else // P is marked as a dirty page in DT
19             DT.SaveDirtyPage(P, L[I_f].LSN);
20 T ← id's of currently active transactions;
21 SaveCheckpointRec(LogFile, DT, T); // creation of a new checkpoint record
22 SaveLogRec(LogFile, "End of a New Checkpoint");
23 Update the master log record such that it points to this newly created checkpoint record;
```

Figure 4. The algorithm for making a checkpoint. During this checkpointing time, committed log records are deleted from the online log table.

The log compaction on page *P* arises in two different ways, depending on the existence of uncommitted online log records of it. If the online log records of *P* are all committed ones, *P* is updated with its current in-buffer image and set to a clean page in lines 10-11. If *P* has one or more uncommitted log records, the algorithm reads page *P* again and applies the committed redo log records to *P* as in lines 13-14. Then, the after-redo image of *P* is written and its PFAT entry is modified as in lines 15-16. In line 17, the log records applied on *P* are deleted from the online log table. Finally, in lines 21-23, the dirty page table and information about in-progress transactions are logged in storage.

4.2. Reincarnating the Buffer Pool

In this section, we address how to restore the buffer pool after system failure. After rebooting the failed system, our recovery module reads the system master record to locate the last checkpoint record in the log file. Then, the actions for buffer reincarnation are conducted according to the recovery algorithm of Figure 5. Through the steps, we can restore the crashed online log table and also build the PFAT data correctly. Recall that all the log records found in the log file are committed ones

because we selectively store only the committed log records rather than uncommitted ones.

Algorithm 2: Procedure *ReincarnateBufferPool*

input: *LogFile* = file pointer to the in-SSD log file;

```
1  CreateEmptyBuffer(Frames, DT, Online_Log_Table); // initialization
2  DT ← LoadDirtyPageTable(Master.CheckpointLoc);
3  RedoStart ← ∞; // setting of the redo starting point
4  for page P ∈ DT do // for each dirty page in DT
5      if P.LSN < RedoStart then RedoStart ← P.LSN;
6  SetFilePtrLoc(LogFile, RedoStart); // rewind the file pointer
7  while (R_log ← GetNextLog(LogFile)) ≠ NULL do // Log scanning
8      P ← R_log.PageId; // processing of a log record for page P
9      if LSN of R_log < LSN of DT and DT.Exist(P) ≠TRUE then
10         continue; // ignore this log record
11     if Frames.Exist(P) ≠ TRUE then
12         Frames.ReadPage(P); // buffer loading of page P
13     if LSN of R_log > Frames.GetLSNInFrame(P) then
14         Online_Log_Table.AddLogRec(R_log); // Log record insertion
15 Frames.FreeAll(); // free the frames allocated to dirty pages
16 foreach page P having its log records in Online_Log_Table do
17     P's PFAT entry ← (-1, dirty); // setting for on-the-fly redos
18 Take a new checkpoint for saving above redo actions and restart the system
   for accepting new transactions;
```

Figure 5. The algorithm for reincarnating the failed buffer pool. This algorithm is used to rebuild the online log table and the PFAT efficiently.

By reading the saved dirty page table, the recovery algorithm finds the redo starting point in lines 4-5. That is, the earliest LSN of the recovery log records is selected as the redo starting point. From the redo starting point, our algorithm begins the redo phase, aiming at restoring the online log table. During the redo phase in lines 7-14, the algorithm reads each committed log record for page *P* and checks if this log record is required for the on-the-fly redo on *P* in line 13. If needed, then the log record is added into the online log table in line 14.

More specifically, our recovery algorithm reads each log record R_{log} and compares its LSN with the LSN of the last checkpoint record. From this, we can determine the time order between R_{log} and the last checkpoint time. If R_{log} is older and is not found in the dirty page table, then we understand that R_{log} was deleted prior to the last checkpoint time. In this case, the update in R_{log} has already been reflected on the database during previous checkpointing. As a result, we can skip record R_{log} as in lines 9-10. In other cases, we read the page *P* associated with R_{log} in lines 11-12. Then, we have to check if the update recorded in R_{log} is reflected on *P* via the LSN comparison. If the LSN of *P* precedes that of R_{log}, then R_{log} is added into the online log table in line 14. After the redo phase during the steps in lines 7-14, the recovery algorithm frees all the buffer frames and updates the PFAT entries of the dirty pages with (-1, dirty) in line 17. From now on,

the correct images of the dirty pages are generated by on-the-fly redo actions during normal database processing. Since there is no need of undo actions in our recovery procedure, we can restart the system after a new creation of the checkpoint record in line 18.

From the recovery mechanism above, we can have two advantages. First, the time for the redo phase is significantly shortened. In the traditional recovery algorithms including ARIES, the redo phase accounts for a large portion of the total recovery time. That is because the traditional algorithm needs to repeat all the lost updates, including uncommitted updates that will be rolled back during the undo phase later. If the portion of such lost updates becomes large because of long checkpoint intervals, then the prolonged redo time could be problematic. Unlike such a traditional redo mechanism, our one requires *no* actual update operations. Instead, we only have to reload the online log from the log file. The actual updates arise through on-the-fly redos during normal database processing. Thus, the delay time needed for redo actions is amortized over the normal transaction processing times. Second, our buffering scheme prevents uncommitted transactions from affecting the databased state. For this, flushing of dirty pages is allowable only at the checkpoint times. Therefore, our recovery procedure does not require the undo phase, thereby shortening the recovery time. Thanks to those two advantages in the recovery procedure, our algorithm can ensure fast system restart, even while checkpoints are being taken with long time intervals. This makes our buffering scheme not enforced to flush dirty pages for the consideration of the fast recovery. Consequently, we can reduce the amount of expensive page updates significantly.

5. Performance Evaluation

In this section, we examine the performance advantages of the proposed buffering scheme. Via extensive experiments based on the TPC-C benchmark [25], we verify that the proposed scheme provides better throughput than the traditional buffering scheme due to the significantly reduced number of page updates.

5.1. Parameter Settings

As the proposed buffering scheme requires a memory-resident log table for the efficient execution of the on-the-fly redo, we need to arrange how to calibrate the memory allocation ratio of the online log table to the total memory space available for the buffer pool. Here, the buffer pool includes the memory areas used for its buffer frames and online log table. To examine the effect of the ratio of the online log table size to the total memory size, we executed the TPC-C benchmark while varying the size of memory space for the online log table. For our experiments, we modified the open-sourced Berkeley DB [26] so that the proposed buffering scheme and the checkpointing algorithm are both incorporated into it.

The experimental results are depicted in Figure 6. Here, we executed 10K transactions according to the TPC-C benchmark scenario and measured the total processing time of 10K transactions. For the experiments, we changed the memory size of the buffer pool to 20MB, 40MB, and 60MB. Within any of the buffer pool sizes, we

changed the size of the online log table incrementally. In the figure, the graphs of the total processing time have the shapes of convex curves. Specifically, in the case of the buffer pool size of 60MB (or 40MB) our scheme yields the best performance when the online log table consumes about 18MB (or 12MB) of memory space. That is, around 30% of the buffer pool size seems to be optimal for the usage of the online log table in those cases. If the memory usage of the online log table exceeds that ratio, it begins to affect adversely the system performance because of reduced numbers of the buffer frames. Note that the memory cannibalization from the buffer frames incurs a greater number of buffer misses, causing performance degradation.

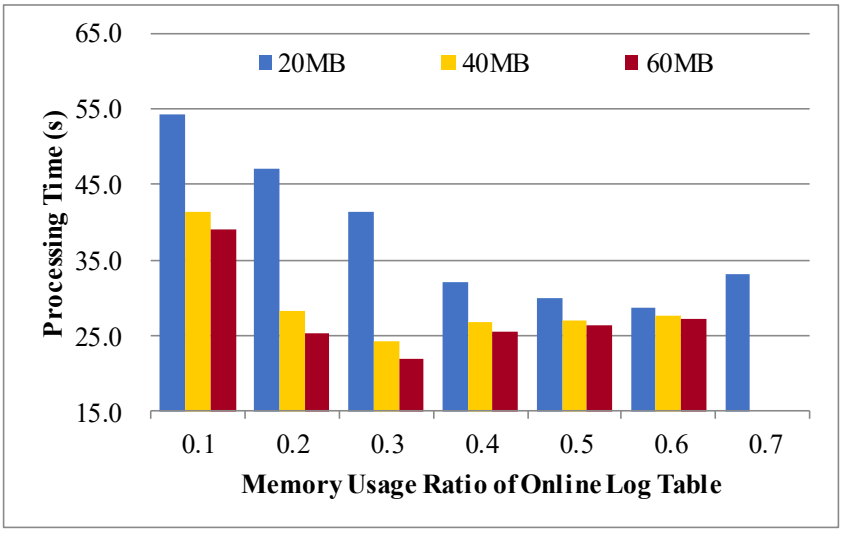


Figure 6. TPC-Benchmark results to show how the transaction processing time varies with respect to an increasing ratio of memory usage of the online log table.

On the other hand, in the case of the smallest buffer pool size of 20MB, we obtain the best performance at 50% to 60% of the memory usage ratio for the online log table. From these observations, we set the lower bound of main memory usage for the online log table as 10MB. Based on the results in Figure 6, we adjusted the size of the online log table in the range of 10MB to 20MB, while performing the TPC-C benchmark in Section 5.2.

As stated in Section 3.2, the group-commit protocol is useful for improving the storage utilization and reducing the amount of log writes. Recall that the committed log records are collected in the log buffer and are written to storage at once. Since the group-commit protocol used for the better logging performance can prolong the average processing time of transactions, it is desirable to maintain a proper balance between the logging efficiency and the delayed processing times.

For this reason, we conducted more experiments in order to get the knowledge about how much space of the log buffer needs to be filled before its flushing. That is, we performed the TPC-C benchmark with 10K transactions, while varying the log file utilization, which is defined as the ratio of the flushed log size with respect to the log buffer size. Figure 7 shows the results. We see that about 80% of the log file utilization

yields the best performance. From the observation, we delay the commits of completed transactions until they fill more than 80% of the log buffer. To prevent excessive delay times of transaction commits, we make the group-commit protocol have a maximum delay time for transactions' commits.

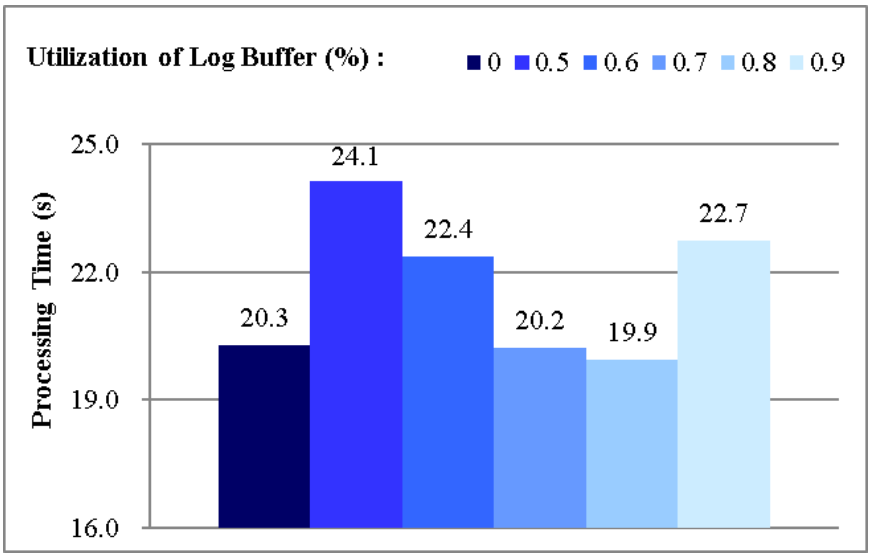


Figure 7. TPC-Benchmark results to show how the transaction processing time varies with respect to the minimum utilization of the log buffer.

5.2. Results and Analyses

To examine the performance improvement with our method, we performed the TPC-C benchmark over the modified Berkeley DB equipped with our scheme and the original Berkley DB. The benchmark results were obtained by executing 100K transactions. To verify that the proposed method works well for ordinary flash storage devices, we picked an off-the-shelf SSD that has the H/W specifications of Table 1. For our experiments, we set the sizes of a data page and a log file page to 8KB and 2KB, respectively. These unit sizes have been widely used in previous literature [17][26]. The database size and the main memory size available for the buffer pool were also chosen according to the environmental specifications of the TPC-C benchmark [25].

To compare the performances between the original Berkeley DB (*'Original'*) and the modified Berkeley DB with our flash-aware buffering scheme (*'Proposed'*), we measured the processing times to complete 100K transactions. The experimental results are given in Figure 8, where the values on the *x*-axis represent the memory size provided for the buffer pool. In the case of *'Proposed'*, we allocate a portion of the total memory capacity for the use of the online log table and the other portion for the buffer frames. The allocation ratios are chosen according to the observations in Section 4.1. In

the case of *'Original'*, we allocate all the memory capacity for buffer frames because it does not use the online log table.

Table 1. SSD Hardware Specs for the TPC-C benchmarks.

CPU	Inter Core 2 Duo 2.53GHz
Used SSD	Samsung 830 Series (128GB)
OS	Solaris 11 (4GB Memory)
Database Size	About 1GB
Data Page Size	8KB
Log Page Size	2KB

In Figure 8, we observe that *'Proposed'* reduces the processing time by 20% to 53%. The improvement mainly comes from the reduction of page updates on storage. In particular, with 60MB of the buffer pool size, *'Proposed'* using the group-commit protocol yields the best performance. In the figure, it is shown that the group-commit protocol employed in *'Proposed'* improves the processing time by 15% up to 27%, compared with the ordinary immediate-commit protocol. This is because the group-commit protocol decreases the overall I/O time in writing log data considerably.

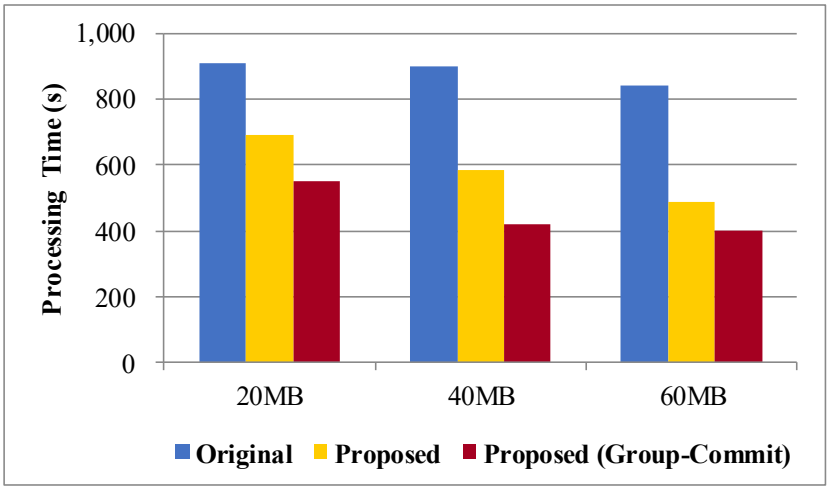


Figure 8. TPC-C Benchmark results to show the performance improvement by using the proposed buffering scheme.

To explore the major causes of the performance improvement, we counted the numbers of page reads and page writes (including page updates) that have been performed during the experiments of Figure 8. Here, the I/O's for saving log data was counted as the number of page writes. Figure 9 depicts the I/O's counting for page reads and writes found in the experiments in Figure 8. As shown in that figure, *'Proposed'* has a greater number of page reads, compared with *'Original'*, while it reduces the number of pages writes. Since the I/O increase in page reads is larger than the decrease in page writes, the proposed scheme could impair the system performance in case of

HDD storage. However, *'Proposed'* ensures better performance as in Figure 8. As easily inferred, such performance enhancement comes from the severe asymmetry between read and write costs in flash storage.

When it comes to the relatively larger workloads of page reads in *'Proposed'*, we can consider two main reasons. First, each flushing-less eviction inevitably incurs one more page read later. Since the mechanism of the on-the-fly redo requires re-caching of any dirty page having flushing-less eviction, *'Proposed'* could not evade an increase in page reads, while providing the flush-less evictions that lead to less page writes. Second, *'Proposed'* may usually experience more buffer replacements during normal database processing. Since a portion of main memory is reserved for the use of the online log table, there is less memory space available for buffer frames in *'Proposed'*. This leads to more buffer misses in *'Proposed'*. As a result, *'Proposed'* produces a more number of page reads than 'Original' as shown in Figure 9.

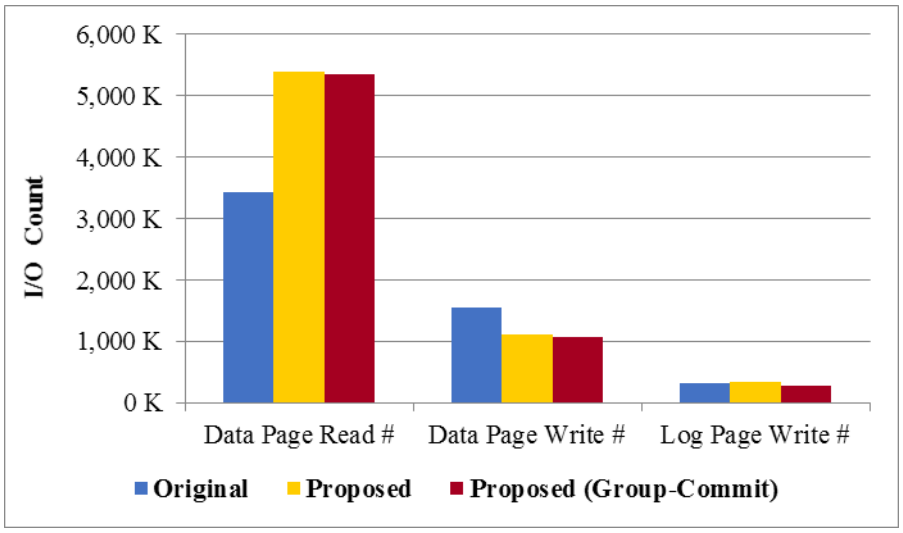


Figure 9. The number of I/Os performed in running the TPC-C Benchmark.

Although *'Proposed'* suffers from lager workloads of page reads during normal database processing, it can diminish the amount of page writes. In Figure 9, more than 30% of page writes (including updates) are reduced via our buffering scheme, compared with the traditional buffering scheme. This is because it does flushing-less evictions of dirty pages and performs restrictive flushing of dirty pages during the checkpoint times. On the other hand, in the respect of the number of log page writes, *'Proposed'* with the immediate-commit protocol works worst as in Figure 9. This is due to the smallest sizes of log data written through a single I/O. In the case of *'Proposed'* using the group-commit protocol, however, the number of log page writes is reduced a lot. Since *Proposed'* stores redo log data only (i.e., no undo log data), it can reduce the total size of log data. Additionally, it stores only the committed log records (i.e., no undo log records). Because of less log data stored and deferred log writes, *'Proposed'* with the group-commit protocol can decrease the total workload for storing log data as in Figure 9. From the results shown in Figure 9, we can say that the reduced I/O cost of page

writes fully compensate the enlarged cost of page reads thanks to very cheap page reads in flash storage.

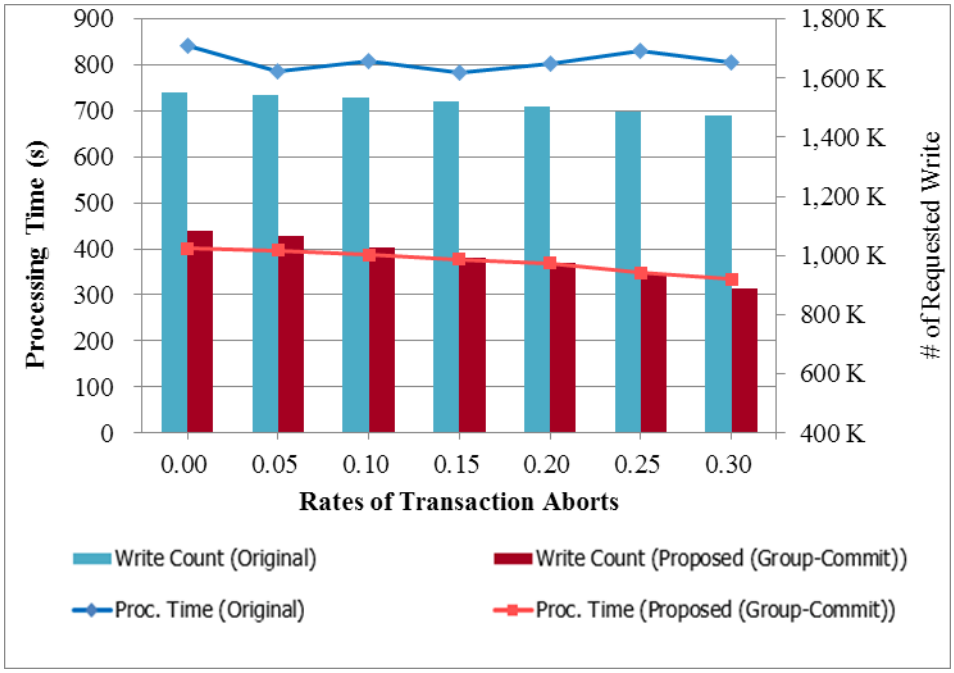


Figure 10. TPC-Benchmark experiments to show how much the transaction processing time varies with respect to an increasing rates of transaction aborts.

When a transaction needs to be aborted for some reasons, the traditional buffering scheme conducts undo actions for rolling back the update operations of the transaction. At the same time, the CLRs are written for preventing the repetitive undo actions at the presence of consecutive system failure. Unlike that, our scheme simply throws away the dirty pages containing these aborted updates from the buffer pool, as stated in Section 3.3. If any of the evicted dirty pages is requested after the transaction's abortion, then our scheme has to read that page again for buffering. Note that, in the case of the traditional buffering scheme, there is no need for reading that page again because it remains in the buffer pool.

For this reason, our scheme may suffer from increasing I/O costs for reloading pages in the presence of frequent transactions aborts. In this concern, we inspect how much the individual transaction aborts affect the performance of the proposed buffering scheme. We measured the number of page writes and the processing time when running 100K transactions, while varying the abortion rates of transactions. Unlike our concern, as in Figure 10, *'Proposed'* shows better adaptation to the tough situations with high transactions' abortion rates, compared with *'Original'*. Contrariwise, *'Proposed'* even yields a somewhat better performance at the high rates of transaction aborts. This is mainly because *'Proposed'* does not write CLRs that are needed in the traditional scheme of *'Original'*. Additionally, *'Proposed'* does not perform undo actions in rolling

back the aborted updates. Since the cost of page reads is very cheap in flash storage, *'Proposed'* seems to provide robust performance improvement in face of frequent transactions aborts.

6. Conclusions

In the case of the traditional buffering scheme in disk-based DBMSs, it is not meaningful to make intense effort for reducing the amount of page updates at the expense of increasing page reads. This is because there exists no significant difference between the I/O costs of a page update and a page read in HDD storage. However, if we attempt to use flash-based devices as primary storage media, such a traditional assumption is not true because of their highly expensive update costs. Therefore, there is a need for developing a new buffering scheme that can efficiently diminish the occurrences of page updates.

To this end, we have proposed a new flash-aware buffering scheme and its recovery algorithm. In our approach, the page-granularity on-the-fly redo is used to enable the flushing-less evictions of dirty pages. Thanks to the on-the-fly redo mechanism, we can easily delay the time of page updates until their owner transactions commit, thereby reducing update operations. Besides the advantage of less page updates, the proposed recovery algorithm guarantees the rapid recovery at the time of system failure because of its undo-free recovery and a very fast redo-phase. In our scheme, it is not required to periodically flush dirty pages during normal database processing for the purpose of the fast recovery.

Despite the advantages above, the proposed buffering scheme needs to allocate some portion of main memory for the on-line-log table. Therefore, our method may suffer from less buffer hit rates because of less memory space assigned for buffer frames. However, such shortcoming can be compensated by the I/O performance gains obtainable from the reduced page updates in our method. To verify this, we performed quite extensive experiments based on TPC-C benchmarks by making the Berkeley DB equipped with our scheme. From the experimental results, we observed that the proposed scheme provides better performance, even though it has a smaller number of buffer frames available. We also verified that the proposed method works well even in the system environment with frequent transaction aborts.

Acknowledgement. This work was supported by (1) a Semiconductor Industry Collaborative Project between Hanyang University and Samsung Electronics Co. Ltd., (2) the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIP) (NRF-2017R1A2B3004581), and (3) the Ministry of Science, ICT and Future Planning (MSIP), Korea, under the Information Technology Research Center (ITRC) support program (IITP-2017-2013-0-00881). Professor Sang-Wook Kim is the corresponding author.

References

1. Leventhal, A.: Flash storage memory. Communications of the ACM, 51(7), 47-51. (2008)
2. Baumann, S., Nijs, G., Strobel, M., and Sattler, K.: Flashing databases: expectations and limitations. In: Proceedings of Data Management on New Hardware. (2010)

3. Na, G., Lee, S., and Moon, B.: Dynamic in-page logging for B+-tree index. IEEE Transactions on Knowledge and Data Engineering 24(7), 1231-1243. (2012)
4. Ganim, M., Mihaila, G., Bhattacharjee, B., Ross, A., and Lan, C.: SSD buffer pool extensions for database systems. In: Proceedings of the International Conference on Very Large Data Bases. pp. 1435-1446. (2010)
5. Do, J., Zhang, D., Patel, J, and DeWitt, D.: Fast peak-to-peak restart for SSD buffer pool Extension. In: Proceedings of the International Conference on Data Engineering. pp. 1129-1140. (2013)
6. Li, Y., He, B., Yang, R., Luo, Q., and Yi, K.: Tree indexing on solid state derives. In: Proceedings of the International Conference on Very Large Data Bases. pp. 1195-1206. (2010)
7. Park, S., Song, H., and Lee, D.: An efficient buffer management scheme for implementing a B-tree on NAND flash memory. In: Proceedings of International Conference on Embedded Software and Systems, (2007)
8. Agrawal, D., Ganesan, D., Sitaraman, R., Diao, Y., and Singh, S.: Lazy adaptive tree: an optimized index structure for flash devices. In: Proceedings of the International Conference on Very Large Data Bases. pp. 361-372. (2009)
9. Lee, S., Moon, B., and Park, C.: Advances in flash memory SSD technology for enterprise Database applications. In: ACM SIGMOD International Conference on Management of Data. (2009)
10. Bae, D., Chang, J., and Kim, S.: An efficient method for record management in flash memory environment. Journal of Systems Architecture 58, 221-232. (2012)
11. Lee, S. and Moon, B.: Design of flash-based DBMS: an in-page logging approach. In: ACM SIGMOD International Conference on Management of Data. pp. 55-66. (2007)
12. Lee, S. and Moon, B.: Transactional in-page logging for multiversion read consistency and recovery. In: Proceedings of the International Conference on Data Engineering. pp. 876-887 (2011)
13. Mohan, C., Lindsay, B., Pirahesh, H., and Schwarz, P.: ARIES: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. ACM Trans. on Database Systems 17 (1), 94–162. (1992)
14. O'Neil, P., Cheng, E., Gawlick, D., and O'Neil, E.: The log-structured merge tree (LSM-Tree). Acta Informatica 33(1), 351-385. (1996)
15. Jeong K., Kim, S., and Lim, S., "A flash-aware buffering scheme using on-the-fly redo," In: Proceedings of the International Conference on Information and Knowledge Management. pp. 1683-1686. (2015)
16. On, S., Li, Y., He, B., Wu, M., Luo, Q., and Xu, J.: FD-buffer: a buffer manager for databases on flash disks," In: Proceedings of the International Conference on Information and Knowledge Management. pp. 1297-1300. (2010)
17. On, S., Xu, J., Choi, B., Hu, H., and He, B.: Flag commit: supporting efficient transaction recovery on flash-based DBMSs. IEEE TKDE 24 (9), 1624-1639. (2012)
18. Do, J., Zhang, D., Patel, J., DeWitt, D., Naughton, J., and Halverson, A.: Turbocharging DBMS buffer pool using SSDs. In: ACM SIGMOD International Conference on Management of Data. (2011)
19. Lee, S., Shin, D., Kim, Y., Kim, J.: LAST: locality-aware sector translation for NAND flash memory-based storage systems. ACM SIGOPS Review, 36-42. (2008)
20. Gupta, A., Kim, Y., and Urgaonkar, B.: DFTL: a flash translation layer employing demand-based selective caching of page-level address mappings," In: Proceedings of ASPLOS, (2009)
21. Moon, S., Lim, S., Park, D, and Lee, S.: Crash recovery in FAST FTL. LNCS 6399, 13-22. (2011)
22. Wu, C., Kuo, T., and Chang, L.: An efficient B-tree layer implementation for flash memory storage systems. ACM Transactions on Embedded Computing Systems 6(3). (2007)

23. Gray, J. and Reuter, A.: Transaction processing: concepts and techniques. Morgan Kaufman Publishers. (1992)
24. Ou, Y., Härder, T., and Jin, P.: FDC: a flash-aware replacement policy for database buffer management. In: Proceedings of Data Management on New Hardware. pp. 15-20. (2009)
25. Oracle, Oracle Berkeley DB, Web page: http://www.oracle.com/technetwork/products/berkeleydb/ (2016)
26. Lee, S., Park, D., Chung, T., Lee, D., Park, S., and Song, H.: A log buffer-based flash translation layer using fully associative section translation. ACM Transactions on Embedded Computing Systems 6(3), 1-27. (2007)

Kyosung Jeong received the MS degree in Electronics and Computer Engineering from Hanyang University, Seoul, Korea, in 2013. He is currently a researcher at Saltlux, an artificial intelligence company. His research interests include databases, knowledge bases, data modeling, and data structure in flash memory.

Sungchae Lim received the BS degree in Computer Engineering from Seoul National University in 1992, and earned the MS and PhD degrees from Korea Advanced Science and Technology (KAIST) in 1994 and 2003, respectively. He is currently an associate professor working for the Dongduk University in Korea. His research interests include flash-memory storage and high-performance indexing schemes.

Kichun Lee worked for ETRI, Tmax Soft, and Samsung SDS from 1999 to 2006. In 2010, he received his Ph.D. in statistics in industrial systems engineering from Georgia Institute of Technology. After postdoctoral associate positions at Georgia Institute of Technology and Emory University, he is now working at the Department of Industrial Engineering, Hanyang University in Seoul, South Korea. He researches data mining, knowledge extraction, and enterprise computing on the domains of information systems and knowledge services.

Sang-Wook Kim received the BS degree in Computer Engineering from Seoul National University, Seoul, Korea in 1989 and earned the MS and PhD degrees from Korea Advanced Science and Technology (KAIST) at 1991 and 1994, respectively. He is Professor at Department of Computer Science and Engineering, Hanyang University, Seoul, Korea. Professor Kim worked with Carnegie Mellon University and IBM Watson Research Center as a visiting scholar. He is an associate editor of Information Sciences. His research interests include data mining and databases.

Received: August 30, 2016; Accepted: January 24, 2017