JOURNAL OF INFORMATION SCIENCE AND ENGINEERING 32, 541-557 (2016)

DOI:10.1688/JISE.2016.32.3.2

A Self-Tuning Buffer-Flushing Algorithm for OLTP Workloads*

KI-HOON LEE[1,+], YUN-GU LEE[2] AND JANGYOUNG KIM[3]
[1]*Department of Computer Engineering*
[2]*Department of Computer Science and Engineering*
Kwangwoon University
Seoul, 01897 Korea
E-mail: [1]*kihoonlee@kw.ac.kr;* [2]*harmony96@gmail.com*
[3]*Department of Computer Science*
University of Suwon
Suwon, 18323 Korea
E-mail: jykim77@suwon.ac.kr

These days, huge amounts of data are created by not only humans but also the Internet of Things (IoT). As databases increase in size, their buffer pool is becoming more crucial to the overall performance. Buffer flushing, *i.e.*, writing dirty pages, is a key factor in buffer pool management. Tuning the buffer-flushing activity in a real system is often difficult because it depends on the workload and system configuration. It is usually done through trial and error. In this paper, we propose a self-tuning algorithm for automatically controlling the buffer-flushing activity. Our algorithm not only avoids synchronous writes, which block other operations, but also preserves the effectiveness of the buffer pool. Experimental results using TPC-C and TPC-H show that our algorithm gives up to a 2.1-fold higher throughput compared with the original buffer-flushing algorithm of MySQL and 1.5-fold compared with that of Percona Server, which is a variant of MySQL.

Keywords: buffer management, buffer flushing, self-tuning algorithm, OLTP, MySQL

1. INTRODUCTION

The buffer pool caches disk pages to reduce expensive I/O operations. This is important to the performance of any database, especially for a big database whose size is much larger than that of the main memory. There has been a lot of work on buffer replacement policies such as LRU, which determine which of the buffer pages to be evicted from the buffer pool. However, so far, little attention has been paid to another important issue in buffer management: *buffer flushing*. Buffer flushing is the activity of writing dirty pages to the disk.

The rate of buffer flushing can affect the system throughput significantly under I/O-bound workloads [1]. If it is too slow, synchronous writes, which cause a waiting period for I/O completion, will occur owing to a lack of free pages in the buffer pool or free space in the redo log files. If it is too fast, the effectiveness of the buffer pool will be

Received June 20, 2015; revised November 19, 2015; accepted November 23, 2015.
Communicated by Vincent S. Tseng.
* This research was supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Science, ICT & Future Planning (NRF-2015R 1C 1A 1A02036517). The present Research has been conducted by the Research Grant of Kwangwoon University in 2015.
+ Corresponding author.

reduced, resulting in more I/O operations. Owing to the complexity of real systems, the manual tuning of the buffer-flushing rate is difficult and time-consuming because it depends on the workload and system configuration (for example, whether the database is stored on HDD or SSD devices). Furthermore, when the workload or configuration changes, the tuning must be performed again. A partial solution [1] for this problem has been proposed for IBM DB2, which enables us to dynamically control the buffer-flushing rate. However, the solution considers only one type of synchronous writes and is not self-tuning because it introduces new tuning parameters.

In this paper, we propose a self-tuning buffer-flushing algorithm that minimizes all types of synchronous writes without harming the effectiveness of the buffer pool. Furthermore, our algorithm does not introduce any new tuning parameters. Our algorithm increases the rate of buffer flushing when synchronous writes occur and decreases the rate when we flush pages beyond the I/O capacity of the system or unnecessarily flush more pages than required to eliminate synchronous writes. One of the challenging issues is that there are other activities running concurrently with the buffer-flushing activity on the same system. These other activities could accidentally overload the I/O subsystem and make us hard to detect whether the current rate of buffer flushing is beyond the I/O capacity or not. We solve this issue using a notion of a *stable state*, which represents a time period where the rate of buffer flushing is not needed to be adjusted. If the number of pages flushed for the current time period is less than or equal to that of the last stable state, we can know that the current rate of buffer flushing is not too fast.

We conducted a detailed experimental evaluation using TPC-C [2] and TPC-H [2]. TPC-C is an OLTP workload, which is write-intensive, and TPC-H is a Decision Support System (DSS) workload, which is read-intensive. We implement our algorithm by extending the existing buffer-flushing algorithm of MySQL [3]. The experimental results show that our algorithm significantly outperforms commercialized methods. Our algorithm improves the throughput up to 2.1-fold compared with the original algorithm of MySQL and 1.5-fold compared with that of Percona Server [4], which is a variant of MySQL. Although our algorithm is focused on MySQL, we believe that it is an important and significant work because MySQL alone is used by millions of users including Facebook, Twitter, LinkedIn, and Adobe.

The remainder of this paper is organized as follows. Section 2 introduces the existing buffer-flushing algorithms, and Section 3 reviews other existing works. Section 4 proposes a new self-tuning algorithm for buffer flushing. Section 5 provides an experimental evaluation of our algorithm. Finally, Section 6 provides some concluding remarks.

2. BACKGROUND

2.1 Buffer Management Overview

MySQL maintains three lists for the buffer pool: an LRU list, a free list, and a flush list. The LRU list stores clean and dirty pages in the order of the last access to the page. Pages evicted from the LRU list are moved to the free list. Thus, the free list has pages that can be used immediately. The flush list maintains only dirty pages in the order in which they were made dirty to preserve the data durability. If a page is modified during a

transaction, the page is inserted into the flush list at the commit stage.

In buffer flushing, there are two types of writes: *replacement writes* and *recoverability writes* [5, 6]. Replacement writes are issued to evict dirty pages from the LRU list and make free pages for reads. Recoverability writes are issued to write dirty pages from the flush list before the redo log files are out of space. Redo log files are used in a circular fashion. To overwrite old redo log entries, the dirty pages corresponding to these entries should be written to the disk first. Recoverability writes also reduce the recovery time by reducing the number of redo log entries to be replayed. We note that a recoverability write of a dirty page does not lead to an eviction of the page from the LRU list.

To improve the performance, the buffer manager of modern DBMSs does not immediately write modifications to the disk. Delaying writes allows user threads (sessions) to not be blocked by writes and continue with their execution. It also allows us to absorb multiple updates on a page before writing the page to the disk. This is called *write combining* (or *write merging*). The buffer manager asynchronously writes dirty pages using a *page cleaner* [1, 6], which is a background thread. The page cleaner periodically collects dirty pages and issues replacement writes to secure free pages in advance. It also issues recoverability writes roughly at the rate of redo log generation so that there will be no dirty pages waiting to be flushed at the time we overwrite old redo log entries. This heuristics for recoverability writes is called *adaptive flushing* [3].

If the page cleaner does not flush pages fast enough, *synchronous writes* occur, which seriously degrade the performance. There are two cases. First, there is no free page in the buffer pool, and a dirty page should be flushed from the LRU list first before the user thread reads a disk page. This incurs *synchronous replacement writes*. Second, the redo log space is full, and dirty pages should be flushed from the flush list first before any data manipulation activity takes place. This incurs *synchronous recoverability writes*.

2.2 The Buffer-Flushing Algorithm of MySQL

Algorithm 1 shows the buffer-flushing algorithm of MySQL. Every one second, it flushes pages from the LRU and flush lists and sleeps if needed to make the buffer-flushing activity periodic. In line 5, $T_{flushing}$ denotes the buffer-flushing time for each iteration. We will explain P_{scan} and P_{io} later.

Algorithm 1: Buffer-flushing algorithm of MySQL
1. $P_{scan} \leftarrow 1024$, $P_{io} \leftarrow 200$
2. **loop**
3. flush pages from the LRU list with the parameter P_{scan}
4. flush pages from the flush list with the parameter P_{io}
5. sleep($1 - T_{flushing}$)
6. **end loop**

For the buffer flushing from the flush list in line 4, MySQL uses the adaptive flushing heuristics to have sufficient free redo log space and a fast recovery time. The heuristics calculates the number of pages to flush based on factors such as the percentage of dirty pages in the buffer pool, the amount of free redo log space, and the rate at which the redo log is generated.

As you can see in lines 3 and 4, there are two critical parameters significantly affecting the rate of buffer flushing: the larger the value the faster the buffer-flushing rate. The parameters are `innodb_lru_scan_depth` (or, P_{scan}) for the LRU list and `innodb_io_capacity` (or, P_{io}) for the flush list.

P_{scan} is the number of free pages that MySQL attempts to keep by periodically evicting pages from the LRU list. If there is no free page at the time the user thread reads a disk page, MySQL scans the LRU list to the P_{scan} depth to find a clean page to evict. If it cannot find a clean page, it now finds a dirty page to evict and synchronously writes the page. We should properly set P_{scan}. If P_{scan} is too high, we have more buffer cache misses and less write combining, which incur more I/O operations. If it is too low, synchronous replacement writes occur.

On the other hand, P_{io} is the I/O capacity available for background tasks such as the adaptive flushing. P_{io} is specified in the number of I/O operations per second (IOPS). The upper limit of P_{io} is defined by the `innodb_io_capacity_max` parameter, which limits the maximum number of pages that can be flushed from the flush list per second. If P_{io} is too high, we have less write combining, which incurs more I/O operations. If it is too low, synchronous recoverability writes occur.

In MySQL, the default values of P_{scan} and P_{io} are 1,024 and 200, respectively. The `innodb_io_capacity_max` parameter is set to twice of P_{io}, with a lower limit of 2,000. Users should manually adjust these parameters according to the workload and system configuration as they are not automatically tuned. For example, suppose that we have SSDs offering much higher IOPS than the default value of P_{io}. The user should increase P_{io}; otherwise, the rate of recoverability writes is limited by P_{io}, but not by the IOPS of SSDs.

2.3 The Buffer-Flushing Algorithm of Percona Server

Algorithm 2 shows the buffer-flushing algorithm of Percona Server, which is a variant of MySQL. When synchronous writes are expected in the near future, Percona Server reduces or skips the sleep of the page cleaner to flush more pages. In line 1, T_{LRU} is the sleep time for the LRU list flushing, and T_{flush} is the sleep time for the flush list flushing. T_{LRU} is reduced if the free list is nearly depleted. In line 5, T_{LRU} is determined as follows [4]: if the free list is filled less than 1% of the buffer pool size: no sleep (0); less than 5%: 50ms shorter sleep time (–50ms) than the previous iteration; between 5% and 20%: no change; more than 20%: 50ms longer sleep time (+50ms). In line 6, T_{flush} is set to zero if the redo log space is full; otherwise, T_{flush} is set to one second.

Algorithm 2: Buffer-flushing algorithm of Percona Server

1. $P_{scan} \leftarrow 1024$, $P_{io} \leftarrow 200$, $T_{LRU} \leftarrow$ 1s, $T_{flush} \leftarrow$ 1s
2. **loop**
3. flush pages from the LRU list with the parameter P_{scan}
4. flush pages from the flush list with the parameter P_{io}
5. adjust T_{LRU}
6. adjust T_{flush}

7. sleep(min(T_{LRU}, T_{flush}) − $T_{flushing}$)
8. **end loop**

Percona Server has also implemented time limits (one second) for the LRU and flush list flushing in lines 3 and 4. If one of the flushing activity takes a long time, it prevents the other kind of flushing activity from running, which in turn may cause many synchronous replacement or recoverability writes depending on the starved flushing activity. This happens when we flush pages beyond the I/O capacity of the system.

3. RELATED WORK

Bridge *et al.* [7] introduced the features of the Oracle buffer manager. Megiddo and Modha [8] proposed a new buffer replacement policy called ARC, which shows a better hit ratio than LRU. On *et al.* [9] and Jiang *et al.* [10] proposed cost-based buffer replacement algorithms for flash memory. Kang, Min, and Eom [11] proposed a buffer replacement scheme for mobile flash storages such as eMMCs and microSD cards. Kim and Lee [12] proposed a novel buffer replacement scheme for flash memory-based portable media players. Jin, Cho, and Chung [13] proposed a buffer replacement policy suitable for the implementation of B^+-tree indices in flash memory. Liu and Salem [6] and Lv *et al.* [14] proposed buffer replacement policies for SSD/HDD hybrid storage systems. Varma and Jacobson [15] and Nam and Park [16] proposed adaptive cache management algorithms for a disk array with a non-volatile write cache. Zhang *et al.* [17] propose an adaptive page migration strategy (APMS) for DRAM/PRAM hybrid memory systems, which reduces migration energy. Yui *et al.* [18] proposed a non-blocking scheme that fixes buffer frames without locks to improve scalability. These efforts are an orthogonal complement to our work. The focus of this paper is not a buffer replacement policy nor specialized cache management algorithms for a particular storage hardware such as disk arrays, flash memory, SSD, and PRAM. We use LRU as replacement policy and focus on autonomously controlling the rate of buffer flushing without specializing to any particular storage hardware.

Wang and Bunt [1] proposed a buffer flushing algorithm for IBM DB2. They consider only synchronous replacement writes and introduce new tuning parameters for the algorithm, and thus their algorithm is not self-tuning. They do not implement the algorithm in DB2 and study the performance using only a buffer pool simulator. The algorithm is not applicable to MySQL because DB2 and MySQL have different buffer pool flushing policies [19]. In contrast, we provide a comprehensive algorithm that avoids synchronous replacement and recoverability writes and does not introduce any new parameters. We implemented our algorithm in MySQL and conducted extensive experiments using industry-standard benchmarks.

The adaptive buffer-flushing heuristics of MySQL prevents a situation in which a free redo log space runs out. Our algorithm also prevents this situation using the same heuristics. Furthermore, our algorithm reduces synchronous writes without incurring unnecessary buffer flush operations.

Percona Server uses an enhanced buffer-flushing algorithm that reduces synchronous writes. However, it introduces new ad hoc constants such as 1%, 5%, 20%, 50ms,

and 1s. These ad hoc constants cannot be dynamically adjusted. Owing to the ad hoc constants, the algorithm could flush more pages than required as we will see in the experiments.

There are a number of studies [19-24] focused on MySQL. Because MySQL is very popular with millions of users, the studies have significant practical value. Furthermore, for researchers who are not the employees or collaborators of the commercial DBMS vendors like Oracle, IBM, and Microsoft, open-source DBMSs like MySQL are the only option that they can implement their algorithm in a DBMS engine. Mozafari *et al.* [19] studied resource and performance modeling focused on MySQL. Lee [20] provided a comprehensive solution to improve the performance of compressed databases for MySQL. Shankar *et al.* [21] explored the potential of employing RDMA to improve the performance of Online Data Processing (OLDP) workloads on MySQL using Memcached for real-world web applications. Cao *et al.* [22] proposed a low cost and high performance MySQL cloud scheme called the UMP (Unified MySQL Platform) system. Tomic *et al.* [23] presented a MySQL storage engine called MoSQL using a transactional distributed key-value store system for atomicity, isolation and durability and a B^+-tree for indexing purposes. Shehab *et al.* [24] presented a formal approach to discover anomalies in database policies using MySQL as case study.

4. SELF-TUNING BUFFER-FLUSHING ALGORITHM

In this section, we propose a self-tuning buffer-flushing algorithm. Section 4.1 explains the design goals. Section 4.2 presents the outline of the algorithm. Section 4.3 presents the algorithm in detail.

4.1 Design Goals

We identify three design goals in order to overcome the drawbacks of the buffer-flushing algorithms of MySQL and Percona Server and to improve overall performance.

1. **Minimizing synchronous writes:** Our algorithm minimizes synchronous writes which seriously degrade the performance.
2. **Minimizing unnecessary I/O operations**: Our algorithm does not unnecessarily flush many pages. Flushing too much harms the effectiveness of the buffer pool and incurs more buffer cache misses and less write combining.
3. **No ad hoc parameters and constants:** Our algorithm does not introduce any new ad hoc parameters and constants to be a true self-tuning algorithm.

4.2 Outline of the Algorithm

To achieve Goal 1, we increase P_{scan} or P_{io} when synchronous writes occur and decrease them when we flush pages beyond the I/O capacity of the system. To achieve Goal 2, we decrease P_{scan} or P_{io} when they are unnecessarily higher than required to eliminate synchronous writes. To achieve Goal 3, we only use statistics that are already available in MySQL. Table 1 summarizes the condition-action rules used in our algorithm to achieve the goals.

Table 1. Condition-action rules.

Condition	Action
synchronous replacement writes occur	increase P_{scan}
synchronous recoverability writes occur	increase P_{io}
P_{scan} is higher than the I/O subsystem can sustain	decrease P_{scan}
P_{io} is higher than the I/O subsystem can sustain	decrease P_{io}
P_{scan} is higher than required	decrease P_{scan}
P_{io} is higher than required	decrease P_{io}

We can check each condition in Table 1 using the information in Table 2, which is already offered by MySQL internally. For example, we can know whether synchronous replacement writes occur or not by checking if $NP_{dirty} > 0$ because NP_{dirty} is the number of pages scanned by user threads to find a dirty page to evict from the LRU list. We can check the occurrence of synchronous recoverability writes using NP_{sync_rec}.

Table 2. Summary of notations.

Symbol	Definition
$T_{flushing}$	buffer-flushing time for each iteration
P_{scan}^{MIN}	the minimum value of P_{scan} defined by MySQL
P_{io}^{MIN}	the minimum value of P_{io} defined by MySQL
NP_{LRU}	the number of pages flushed from the LRU list
NP_{flush}	the number of pages flushed from the flush list
NP_{LRU}^{Stable}	NP_{LRU} at the last stable state
NP_{flush}^{Stable}	NP_{flush} at the last stable state
$\|LRU\|$	the length of the LRU list
$\|free\|$	the length of the free list
NP_{clean}	the number of pages scanned to find a clean page to evict
NP_{dirty}	the number of pages scanned to find a dirty page to evict
NP_{req_flush}	the number of pages requested to flush from the flush list
NP_{sync_rec}	the number of pages written by synchronous recoverability writes

Although it is easy to check if synchronous writes occur, it is challenging to check if whether P_{scan} or P_{io} are too high or not. There are two cases where they are too high.

The first case is that P_{scan} or P_{io} are higher than the I/O subsystem can sustain. We can detect this case by checking if the buffer-flushing activity of the current iteration takes up more than one second (*i.e.*, $T_{flushing} > 1$s) because MySQL was designed under the assumption $T_{flushing} \leq 1$s. However, sometimes checking only $T_{flushing} > 1$s is not enough because other activities could be running concurrently on the same system and they overloaded the I/O subsystem. We solve this problem using the notion of a *stable state*. We define the stable state as the iteration where $T_{flushing} \leq 1$s and P_{scan} or P_{io} is not changed. We record the number of pages flushed at the last stable state. Even if $T_{flushing} > 1$s, if the number of pages flushed for the current iteration is less than or equal to that of

the last stable state, we are not in the first case because the I/O subsystem was accidentally overloaded for the current iteration by other activities.

The second case is that P_{scan} or P_{io} are higher than required. For P_{scan}, we can detect this case by checking if there are surplus free pages left over and user threads do not need to find clean pages by scanning the LRU list. For P_{io}, we can detect this case by checking if P_{io} is higher than NP_{req_flush}, which is the number of pages requested to flush from the flush list by the adaptive flushing heuristics of MySQL.

4.3 Algorithm

Algorithm 3 shows the self-tuning buffer-flushing algorithm that extends Algorithm 1 of MySQL to dynamically adjust P_{scan} and P_{io}. Fig. 1 shows the flow chart of Algorithm 3. It implements the condition-action rules in Table 1. Lines 2, 3, 4, 40, 42 is the same as Algorithm 1 including the adaptive flushing heuristics. Table 2 lists the notations used in the algorithm. The page cleaner periodically flush pages, and we adjust P_{scan} and P_{io} based on information gathered for each iteration. We set the `innodb_io_capacity_max` parameter to twice of P_{io} without a fixed lower limit.

We will now describe the details of our algorithm. In lines 5 to 17, if $T_{flushing} > 1$s, we slow down the rate of buffer flushing by decreasing P_{scan} or P_{io}. This corresponds to the left branch of the flow chart shown in Fig. 1. We determine who takes the majority of buffer flushing in line 7 by comparing NP_{LRU} and NP_{flush}. Let the majority is the buffer flushing from the LRU list, that is, $NP_{LRU} > NP_{flush}$. We compare the number of pages NP_{LRU} flushed for the current iteration and the number pages NP_{LRU}^{Stable} flushed at the last stable state. If $NP_{LRU} > NP_{LRU}^{Stable}$, we flush more pages than at the last stable state, and thus decrease P_{scan} based on the ratio $NP_{LRU}^{Stable}/NP_{LRU}$ to return to a stable state (lines 9 through 11). P_{io} is also adjusted in the same way (lines 14 through 16).

In lines 20 through 39, we adjust P_{scan} and P_{io} when the I/O subsystem is not overloaded (that is, $T_{flushing} \leq 1$s). This corresponds to the right branch of the flow chart shown in Fig. 1. In lines 20 and 22, we increase P_{scan} when synchronous replacement writes occur ($NP_{dirty} > 0$). This case happens when the free list is empty, the user thread cannot find a clean page to evict from the LRU list, and a dirty page to evict is found by scanning NP_{dirty} pages from the LRU list. Thus, we increase P_{scan} by NP_{dirty}. We note that P_{scan} obviously cannot exceed $|LRU|$. On the other hand, an unnecessarily high P_{scan} just increases I/O operations without any benefit. We can know that P_{scan} is unnecessarily high if there are surplus free pages left over ($|free| > 0$) and the user thread does not need to find clean pages ($NP_{clean} = 0$). Thus, we decrease P_{scan} by $|free|$. In lines 26 and 28, if we do not need to adjust P_{scan} for the current iteration, we set NP_{LRU}^{Stable} to NP_{LRU} because it means that the current iteration is in a stable state. Here, NP_{LRU} should be greater than 0 because 0 indicates an idle state rather than a stable state.

In lines 30 through 39, we adjust P_{io}. The adaptive flushing heuristics of MySQL calculates the number of pages NP_{req_flush} requested to flush from the flush list. We use NP_{req_flush} as the yardstick. When synchronous recoverability writes occur ($NP_{sync\_rec} > 0$) and P_{io} is not sufficient to flush NP_{req_flush} pages ($P_{io} < NP_{req\_flush}$), we increase P_{io} by NP_{sync_rec}. In lines 33 and 35, if P_{io} is unnecessarily higher than NP_{req_flush}, we decrease it to NP_{req_flush}. In lines 36 and 38, if we do not need to adjust P_{io} for the current iteration, we set NP_{flush}^{Stable} to NP_{flush} because it means that the current iteration is in a stable state.

Algorithm 3: self-tuning buffer-flushing algorithm

1. $NP_{LRU}^{Stable} \leftarrow P_{scan}^{MIN}$, $NP_{flush}^{Stable} \leftarrow P_{io}^{MIN}$
2. **loop**
3. flush pages from the LRU list with the parameter P_{scan}
4. flush pages from the flush list with the parameter P_{io}
5. **if** $T_{flushing} > 1$s **then**
6. // the I/O subsystem is overloaded
7. **if** $NP_{LRU} > NP_{flush}$ **then**
8. // LRU list flushing is the majority
9. **if** $NP_{LRU} > NP_{LRU}^{Stable}$ **then**
10. $P_{scan} \leftarrow \max(P_{scan} \times NP_{LRU}^{Stable} / NP_{LRU}, P_{scan}^{MIN})$
11. **end if**
12. **else**
13. // flush list flushing is the majority
14. **if** $NP_{flush} > NP_{flush}^{Stable}$ **then**
15. $P_{io} \leftarrow \max(P_{io} \times NP_{flush}^{Stable} / NP_{flush}, P_{io}^{MIN})$
16. **end if**
17. **end if**
18. **else**
19. // $T_{flushing} \leqq 1$s, that is, the I/O subsystem is not overloaded
20. **if** $NP_{dirty} > 0$ and $P_{scan} + NP_{dirty} \leq |LRU|$ **then**
21. // synchronous replacement writes occur
22. $P_{scan} \leftarrow P_{scan} + NP_{dirty}$
23. **else if** $|free| > 0$ **and** $NP_{clean} = 0$ **and** $P_{scan} \geq |free| + P_{scan}^{MIN}$ **then**
24. // P_{scan} is higher than required
25. $P_{scan} \leftarrow P_{scan} - |free|$
26. **else if** $NP_{LRU} > 0$ **then**
27. // stable state
28. $NP_{LRU}^{Stable} \leftarrow NP_{LRU}$
29. **end if**
30. **if** $NP_{sync\_rec} > 0$ **and** $P_{io} < NP_{req\_flush}$ **then**
31. // synchronous recoverability writes occur
32. $P_{io} \leftarrow P_{io} + NP_{sync\_rec}$
33. **else if** $P_{io} > NP_{req\_flush} \geq P_{io}^{MIN}$ **then**
34. // P_{io} is higher than required
35. $P_{io} \leftarrow NP_{req\_flush}$
36. **else if** $NP_{flush} > 0$ **then**
37. // stable state
38. $NP_{flush}^{Stable} \leftarrow NP_{flush}$
39. **end if**
40. sleep$(1 - T_{flushing})$
41. **end if**
42. **end loop**

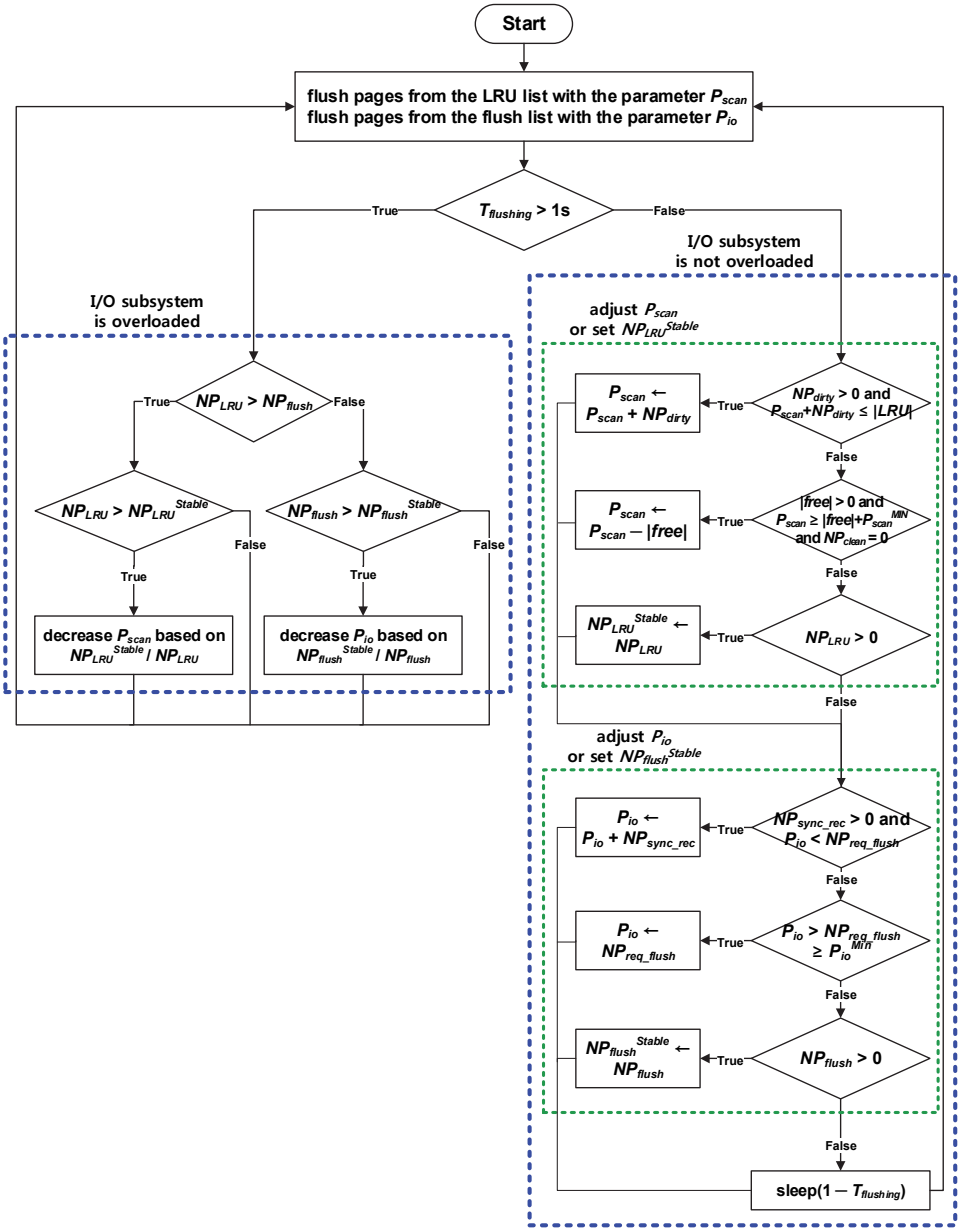

Fig. 1. The flow chart of Algorithm 3.

5. EXPERIMENTAL EVALUATION

5.1 Experimental Setup

We conducted an extensive experimental evaluation of our algorithm using the TPC-C and TPC-H benchmarks. TPC-C is a standard benchmark simulating real OLTP

workloads and write intensive. To measure the maximum performance, we set the key and thinking time to zero for TPC-C. TPC-H is a standard benchmark for decision support systems (DDS) and OLAP and read intensive. For the TPC-C and TPC-H benchmarks, we use Database Test Suite [25, 26] which is a set of open-source implementations of the TPC benchmarks.

We implement our algorithm on MySQL Community Server 5.6.15. The important parameters of MySQL are shown in Table 3. To minimize the interference by data caching at the OS layer, we use direct I/O (`O_DIRECT`). To assure a controlled setting, we do not split the buffer pool or flush the neighbor pages.

Table 3. The MySQL configuration.

Parameter	Configuration
innodb_flush_method	O_DIRECT
innodb_buffer_pool_instances	1
innodb_flush_neighbors	0
innodb_log_file_size	8G
innodb_max_dirty_pages_pct	90%
innodb_thread_concurrency	16

We compare our algorithm, the original algorithm of MySQL (or, the naive algorithm) and the algorithm of Percona Server [4] (or, the Percona's algorithm) in terms of the transaction throughput. We use Percona Server 5.6.15, which has the same version number as MySQL. We measure throughput, the proportion of synchronous writes to all write operations (denoted as $percentage_{sync\_write}$), and the number of I/O operations per transaction. We conduct the following experiments to test various scenarios.

Experiment 1: TPC-C throughput as the buffer pool size is varied We measure transactions per minute (TPM) for TPC-C by exponentially increasing the buffer pool size from 1GB to 8GB. In TPC-C, the size of the database is specified through the scale factor, which is the number of warehouses. We use a large-scale factor of 1,000 (about 100GB) and 20 database connections. We measure TPM after a 30-min warm-up period for SSDs and a two-hour warm-up period for HDDs because HDDs need a longer warm-up period than SSDs.

Experiment 2: TPC-C throughput as the database size is varied We measure TPM for TPC-C by exponentially increasing the size of database from a scale factor 10 to 1000. We set the buffer pool size to 2GB and use 20 database connections. A database with a scale factor of 10 is small enough to fit in the buffer pool.

Experiment 3: TPC-C throughput as the number of connections is varied We measure TPM for TPC-C by exponentially increasing the number of database connections from 1 to 1000. We use a scale factor of 1000 and set the buffer pool size to 1GB.

Experiment 4: TPC-H throughput We measure TPC-H throughput, which is the ratio of the number of queries executed over the measurement interval. We use a scale factor of 1 and 10 query streams and set the buffer pool size to 4GB.

All experiments were conducted on a Linux PC with an Intel Core i7-2600K CPU and 16 GB of main memory. We use Samsung 850 PRO Series 256GB SSDs and Western Digital 1TB HDDs. For all configurations, we used a separate disk for the system table space and logs. All the results are averaged over five runs.

5.2 Experimental Results

5.2.1 Experiment 1: TPC-C throughput as the buffer pool size is varied

Fig. 2 shows the TPC-C throughput in terms of TPM by varying the buffer pool size for SSDs. Compared with the naive algorithm, our algorithm shows a 1.9- to 2.1-fold higher TPM because the $percentage_{sync\_write}$ of the naive algorithm is 8.9% to 9.7%, but that of our algorithm is zero or almost zero (at most 0.04%) as shown in Fig. 3. The $percentage_{sync\_write}$ of the Percona's algorithm is zero for all cases. Compared with the Percona's algorithm, our algorithm shows a 1.2- to 1.5-fold higher TPM because the Percona's algorithm flushes more pages than required, and thus generates more I/O operations per transaction as shown in Fig. 4. In contrast, our algorithm dynamically tunes the number of pages flushed to the minimum number so that only the synchronous writes are eliminated. The number of I/O operations per transaction of our algorithm is greater than that of the naive algorithm because our algorithm flushes more pages to prevent synchronous writes.

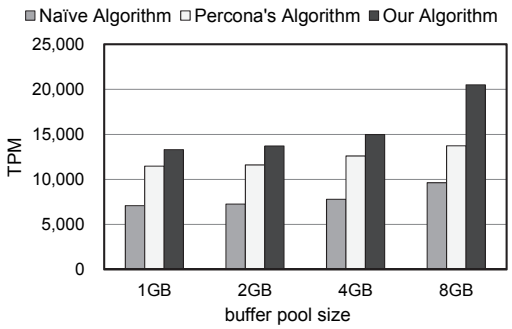

Fig. 2. TPM by varying the buffer pool size for TPC-C on SSDs.

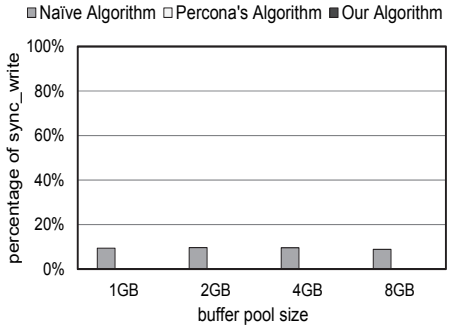

Fig. 3. $percentage_{sync\_write}$ by varying the buffer pool size for TPC-C on SSDs.

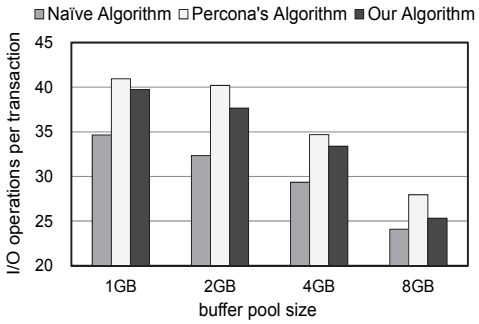

Fig. 4. The number of I/O operations per transaction by varying the buffer pool size for TPC-C on SSDs.

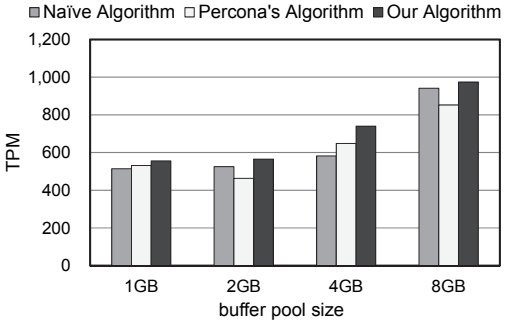

Fig. 5. TPM by varying the buffer pool size for TPC-C on HDDs.

Figs. 5-7 show the results for the HDDs. Our algorithm improves TPM up to 1.3-fold compared with the naive algorithm and 1.2-fold compared with the Percona's algorithm as shown in Fig. 5. The $percentage_{sync\_write}$ of the naive algorithm and the Percona's algorithm is zero as shown in Fig. 6, but they flush more pages than required and generates more I/O operations per transaction as shown in Fig. 7. The $percentage_{sync\_write}$ of our algorithm is not zero but almost zero (0.52% to 1.61%), and the number of I/O operations per transaction of our algorithm is smaller than those of the naive and Percona's algorithms because our algorithm minimizes unnecessary flushing activities. The Percona's algorithm shows irregular and inconsistent performance. For example, for the buffer pool size of 2GB, the Percona's algorithm shows worse TPM than that of 1GB. This is because the Percona's algorithm adjusts the rate of buffer flushing based on ad hoc constants. Because it blindly flushes more pages if the length of the free list, $|free|$, is smaller than a constant percentage C of the buffer pool, it often unnecessarily flushes many pages even when there are surplus free pages left over. For example, suppose that $|free|$ = 80MB and C = 5%. When the buffer pool size is 1GB, the Percona's algorithm does not increase the rate of buffer flushing because 80MB is about 8% of the buffer pool. However, when the buffer pool size is 2GB, it flushes more pages because 80MB is now about 4% of the buffer pool.

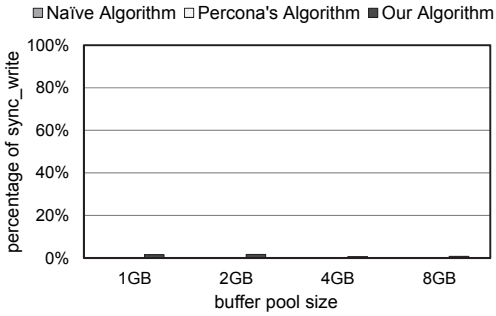

Fig. 6. $percentage_{sync\_write}$ by varying the buffer pool size for TPC-C on HDDs.

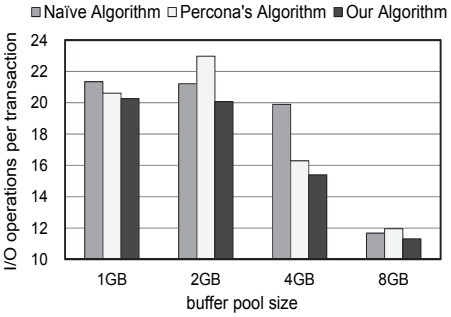

Fig. 7. The number of I/O operations per transaction by varying the buffer pool size for TPC-C on HDDs.

5.2.2 Experiment 2: TPC-C throughput as the database size is varied

Fig. 8 shows TPM by varying the database size for SSDs. When the scale factor is 10, the three algorithms show almost the same TPM because the buffer pool is relatively large, and thus, synchronous writes rarely occur. The naive algorithm slightly underperform our algorithm when the scale factor is 10 because it unnecessarily flushes many pages based on fixed parameter values. We omit the results for HDDs because they show a similar tendency to those of SSDs.

5.2.3 Experiment 3: TPC-C throughput as the number of connections is varied

Fig. 9 shows TPM by varying the number N of database connections for SSDs. As N increases, TPM is increased at first because we can process more requests concurrently

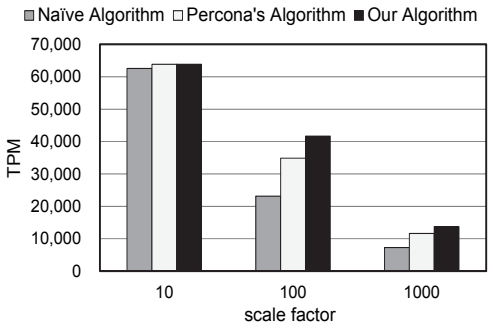

Fig. 8. TPM by varying the database size for TPC-C on SSDs.

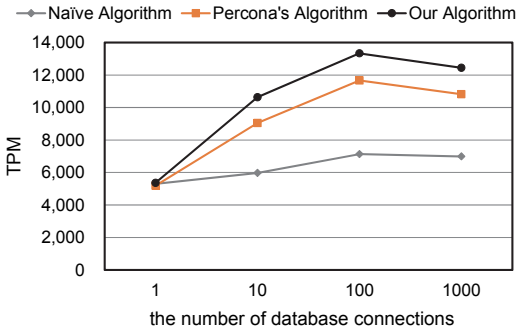

Fig. 9. TPM by varying the number of database connections for TPC-C on SSDs.

and then decreased because of the resource contention and the overhead of connection and queue management. Even when N = 1,000, TPM is not seriously degraded by thrashing because there are a fixed number of internal threads in the DBMS and requests from N connections wait in a queue until a thread is available to process it. Our algorithm outperforms the existing algorithms when $N > 1$ because of the same reason as in Experiment 1. When $N = 1$, all algorithms show almost the same TPM because there is only one thread producing dirty pages. When the amount of dirty pages is small, the rate of buffer flushing does not significantly affect the system throughput. We omit the results for HDDs because they show a similar tendency to those of SSDs.

5.2.4 Experiment 4: TPC-H throughput

Fig. 10 shows TPC-H throughput of the three algorithms on SSDs. They show almost the same throughput because TPC-H is a read-intensive workload, and thus, the rate of buffer flushing is not a significant factor. The Percona's algorithm slightly underperforms other algorithms because it unnecessarily flushes more pages based on ad hoc constants. We omit the results for HDDs because they show a similar tendency to those of SSDs.

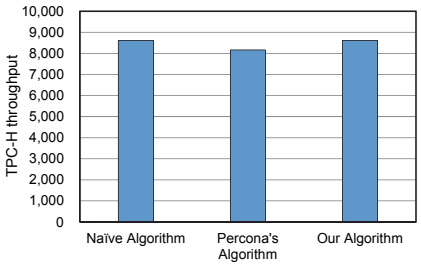

Fig. 10. TPC-H throughput on SSDs.

5.3 Discussions

The naïve algorithm often incurs synchronous writes. The Percona's algorithm is able to avoid synchronous writes, but often unnecessarily flushes many pages even when

there are surplus free pages left over because it adjusts the rate of buffer flushing based on ad hoc constants. Our algorithm significantly outperforms the existing algorithms especially for (1) SSDs because the parameters of the existing algorithms are optimized and fixed for conventional HDDs, (2) a big database whose size is much larger than that of the main memory, (3) more than one concurrent user threads, and (4) a write-intensive workload that generates a large amount of buffer flushing. We believe that our algorithm is also effective for other high performance storages such as high-end HDDs and disk arrays.

6. CONCLUSIONS

Tuning buffer management for the best performance is often difficult to achieve owing to the complexity of real systems. We proposed a novel self-tuning algorithm for buffer management, especially with regard to buffer flushing. Our algorithm dynamically tunes the number of pages flushed to the minimum number so that only the synchronous writes are eliminated. Extensive experiments show that our algorithm substantially outperforms the original algorithm of MySQL and the algorithm of Percona Server.

REFERENCES

1. W. Wang and R. Bunt, "A self-tuning page cleaner for DB2," in *Proceedings of IEEE International Symposium on Modeling*, *Analysis and Simulation of Computer and Telecommunications Systems*, 2002, pp. 81-89.
2. Transaction Processing Performance Council (TPC), http://www.tpc.org.
3. MySQL 5.6 Reference Manual, http://dev.mysql.com/doc/refman/5.6/en.
4. Percona Server 5.6 Documentation, https://www.percona.com/doc/percona-server/5.6.
5. X. Li, A. Aboulnaga, K. Salem, A. Sachedina, and S. Gao, "Second-tier cache management using write hints," in *Proceedings of the USENIX Conference on File and Storage Technologies*, 2005, pp. 115-128.
6. X. Liu and K. Salem, "Hybrid storage management for database systems," in *Proceedings of the VLDB Endowment*, Vol. 6, 2013, pp. 541-552.
7. W. Bridge, A. Joshi, M. Keihl, T. Lahiri, J. Loaiza, and N. MacNaughton, "The oracle universal server buffer," in *Proceedings of the 23rd VLDB Conference*, 1997, pp. 590-594.
8. N. Megiddo and D. S. Modha, "ARC: a self-tuning, low overhead replacement cache," in *Proceedings of the USENIX Conference on File and Storage Technologies*, 2003, pp. 115-130.
9. S. T. On, S. Gao, B. He, M. Wu, Q. Luo, and J. Xu, "FD-buffer: a cost-based adaptive buffer replacement algorithm for flash memory devices," *IEEE Transactions on Computers*, Vol. 63, 2014, pp. 2288-2301.
10. Z. Jiang, Y. Zhang, J. Wang, and C. Xing, "A cost-aware buffer management policy for flash-based storage devices," in *Proceedings of International Conference on Database Systems for Advanced Applications*, 2015, pp. 175-190.
11. D. H. Kang, C. Min, and Y. I. Eom, "Block utilization-aware buffer replacement scheme for mobile NAND flash storage," *IEICE Transactions on Information and*

Systems, Vol. E97-D, 2014, pp.2510-2513.
12. B. Kim and D. Lee, "LSF: a new buffer replacement scheme for flash memory-based portable media players," *IEEE Transactions on Consumer Electronics*, Vol. 59, 2013, pp. 130-135.
13. R. Jin, H.-J. Cho, and T.-S. Chung, "LS-LRU: a lazy-split LRU buffer replacement policy for flash-based B^+-tree index," *Journal of Information Science and Engineering*, Vol. 31, 2015, pp. 1113-1132.
14. Y. Lv, B. Cui, X. Chen, and J. Li, "HAT: an efficient buffer management method for flash-based hybrid storage systems," *Frontiers of Computer Science*, Vol. 8, 2014, pp. 440-455.
15. A. Varma and Q. Jacobson, "Destage algorithms for disk arrays with non-volatile caches," *IEEE Transactions on Computers*, Vol. 47, 1998, pp. 228-235.
16. Y. Nam and C. Park, "An adaptive high-low water mark destage algorithm for cached RAID5," in *Proceedings of Pacific Rim International Symposium on Dependable Computing*, 2002, pp. 177-184.
17. T. Zhang, J. Xing, J. Zhu, and T. Chen, "Exploiting page write pattern for power management of hybrid DRAM/PRAM memory system," *Journal of Information Science and Engineering*, Vol. 31, 2015, pp. 1633-1646.
18. M. Yui, J. Miyazaki, S. Uemura, and H. Yamana, "Nb-GCLOCK: a non-blocking buffer management based on the generalized CLOCK," in *Proceedings of IEEE International Conference on Data Engineering*, 2010, pp. 745-756.
19. B. Mozafari, C. Curino, A. Jindal, and S. Madden, "Performance and resource modeling in highly-concurrent OLTP workloads," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2013, pp. 301-312.
20. K.-H. Lee, "Performance improvement of database compression for OLTP workloads," *IEICE Transactions on Information and Systems*, Vol. E97-D, 2014, pp. 976-980.
21. D. Shankar, X. Lu, J. Jose, M. Wasi-ur-Rahman, N. S. Islam, and D. K. Panda, "Can RDMA benefit online data processing workloads on Memcached and MySQL?" in *Proceedings of IEEE International Symposium on Performance Analysis of Systems and Software*, 2015, pp. 159-160.
22. W. Cao, F. Yu, and J. Xie, "Realization of the low cost and high performance MySQL cloud database," in *Proceedings of the VLDB Endowment*, Vol. 7, 2014, pp. 1742-1747.
23. A. Tomic, D. Sciascia, and F. Pedone, "MoSQL: an elastic storage engine for MySQL," in *Proceedings of the Annual ACM Symposium on Applied Computing*, 2013, pp. 455-462.
24. M. Shehab, S. Al-Haj, S. Bhagurkar, and E. Al-Shaer, "Anomaly discovery and resolution in MySQL access control policies," in *Proceedings of International Conference on Database and Expert Systems Applications*, 2012, pp. 514-522.
25. DBT2 Benchmark Tool, http://dev.mysql.com/downloads/benchmarks.html.
26. Database Test Suite, http://osdldbt.sourceforge.net/.

Ki-Hoon Lee has received B.S. (2000), M.S. (2002), and Ph.D. (2009) degrees in Computer Science from Korea Advanced Institute of Science and Technology (KAIST). He is currently an Assistant Professor of Department of Computer Engineering at Kwangwoon University.

Yun-Gu Lee has received B.S. (2000), M.S. (2002), and Ph.D. (2006) degrees in Electrical Engineering from Korea Advanced Institute of Science and Technology (KAIST). He is currently an Associate Professor of Department of Computer Science and Engineering at Kwangwoon University.

Jangyoung Kim has received his B.S. degree of Computer Science from Yonsei University in Seoul, Korea, M.S. degree of Computer Science and Engineering from Pennsylvania State University in University Park, and Ph.D. of Computer Science and Engineering in University at Buffalo (SUNY). He is currently an Assistant Professor of Department of Computer Science at University of Suwon.