M.S. THESIS

An Efficient Protection of Partial Page Update and Buffer Scheme in a Database System Using Non-Volatile Memory

비휘발성 메모리를 사용한 데이터베이스의 효율적인 부분 페이지 갱신 방지 및 버퍼 기법

BY

Hara Kang

AUGUST 2016

DEPARTMENT OF COMPUTER SCIENCE ENGINEERING
COLLEGE OF ENGINEERING
SEOUL NATIONAL UNIVERSITY

M.S. THESIS

An Efficient Protection of Partial Page Update and Buffer Scheme in a Database System Using Non-Volatile Memory

비휘발성 메모리를 사용한 데이터베이스의 효율적인 부분 페이지 갱신 방지 및 버퍼 기법

BY

Hara Kang

AUGUST 2016

DEPARTMENT OF COMPUTER SCIENCE ENGINEERING
COLLEGE OF ENGINEERING
SEOUL NATIONAL UNIVERSITY

An Efficient Protection of Partial Page Update and Buffer Scheme in a Database System Using Non-Volatile Memory

비휘발성 메모리를 사용한 데이터베이스의 효율적인 부분 페이지 갱신 방지 및 버퍼 기법

지도교수 염 헌영

이 논문을 공학석사 학위논문으로 제출함

2016 년 8 월

서울대학교 대학원

컴퓨터공학부

Hara Kang

Hara Kang의 공학석사 학위논문을 인준함

2016 년 8 월

위 원 장	엄 현상	(인)
부위원장	염 헌영	(인)
위 원	전 병곤	(인)

Abstract

In this paper, we present an efficient approach that exploits persistence, fast data access and byte-addressable properties of non-volatile memory (NVM) to protect partial page update and improve transaction performance on flash SSD-based databases. Our scheme is to unify the protention scheme for partial page update (e.g., doublewrite in InnoDB and full-page write in PostgreSQL) and page buffer scheme for read/write operations with several optimizations using NVM to reduce SSD I/Os and related lock contention. In addition, we place redo log buffer on NVM to remove further I/O overhead for better performance. We implemented our scheme in MySQL InnoDB on an NVDIMM server and improved transaction performance 4X and endurance 3X on OLTP workload compared to the existing scheme.

Keywords: Databases, partial page update, page buffer scheme, non-volatile memory, byte-addressable, redo logging, I/O

Student Number: 2014-22679

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Emerging non-volatile memory (NVM), like phase-change memory (PCM) and spin-transfer torque memory (STT-RAM), has attracted attention in both academia and industry by providing many improved features oover DRAM such as persistence and higher capacity while providing comparable speed of data access (load/store instructions). Accordingly, NVM has been actively investigated as the next-generation memory technology.

To exploit the mentioned features of NVM for improving the database performance, many studies [5, 14, 4, 2] have demonstrated that *Write-Ahead Logging* (WAL) is a critical overhead and improved the performance by removing the overhead. In current databases, Write-Ahead Logging forces log records of a transaction to be flushed to log files on disk before any data changes are made to the data files in order for the recovery process to restore the database to a consistent state [10]. This centralized logging system has become a great inhibitor in performance as it remains as a sequential bottleneck even on massively parallel hardware that are becoming more of a reality [14]. The mentioned

studies [5, 14, 4, 2] have eliminated this I/O overhead and improved the performance and latency to a certain level.

However, as well as logging, we also observed that the technique protecting partial page update in database [6] is another source of significant overhead in SSD-based database system. This is truly so when the database workload is more I/O intensive such as in the case where DRAM buffer pool size is substantially smaller than the total database size. Since page eviction from memory buffer pool will occur much more often in this type of workload, I/O overhead of flushing dirty pages from the buffer pool becomes more substantial compared to the logging overhead, which is quite constant regardless of ratio between memory buffer pool and database size.

A crucial assumption for database consistency and recovery is that individual pages should be written to storage atomically, which is unfortunately not supported by most stable storage devices including disks and flash-bsased SSDs out of the box. If a system crash or a power failure occurs while a page write is in progress, the page may be left with a mix of old and new data. With such a partially written page (i.e., a shorn write), even *Write-Ahead Logging* (WAL) would not be enough to completely restore the consistent state of a database [6]. For this reason, modern database systems provide a strong guarantee against partial update of database pages [9] by redundant page writes, inspired by shadow paging [8]. The InnoDB storage engine of MySQL deals with this problem by utilizing a redundant page update technique called *doublewrite* [13]. However, this scheme significantly affects the database performance on SSD since it performs frequent *write()* system calls for redundant writes and frequent *fsync()* system calls for ordering and durability. PostgreSQL also takes a similar approach to avoid the partial page write problem. When the *full-page write* option is on, PostgreSQL server writes the entire content of a page to

WAL log during the first modification of the page after a checkpoint [12].

In this paper, we propose an efficient solution that unions the protection scheme for partial page update and page buffer scheme for read and write operations by exploiting the persistent and fast data access features of NVM, which makes it an ideal place for the two schemes. Basically, our work challenges the hard problem of how the database page I/O should be changed when NVM is attached to the database system. This is a significant challenge since the solution should provide more efficient page I/O while also guaranteeing the partial page update on secondary storage. Our scheme removes redundant writes to SSD and maximally reduce the frequency of file I/O system calls such as *read()*, *write()*, and *fsync()* that are issued to the underlying file system.

One more issue that might harass SSD-based databases is flash endurance. Despite many benefits that SSD provides, one of the main impediments to the wide adoption of SSDs is its limited write endurance [11]. Due to the characteristics of NAND flash, SSDs have a finite lifetime dictated by the number of write operations known as program/erase (P/E) cycles NAND flash can endure [3]. The intrinsic NAND flash needs to erase blocks before writing to a page, which results in write amplification where the data size written to the physical NAND is in fact several times larger than the size of the data that is intended to be written by the host system. This further aggravates endurance issue of flash-based SSDs, which presents dire need for efficient reduction in total amount of I/Os. By aiming at removing I/Os from the protection technique, we not only increase database performance but also improve flash endurance of the SSDs.

We analyze the doublewrite buffer (DWB) of MySQL InnoDB storage engine as the engine is a widely used ACID compliant transaction manager. We show that applications using MySQL can achieve a significant performance advantage while still maintaining ACID compliance by optimizing InnoDB to

utilize the new NVM storage. Our scheme is implemented and evaluated on a real NVDIMM platform, which is commercially available [1].

The key contributions of our work are:

1. We provide a detailed analysis of the overhead for protection technique of partial page update from real implementation.

2. Based on the observations, we propose an efficient buffer scheme for exploiting emerging NVM technology on databases with flash-based SSD as main storage.

3. Based on our experimental results, we give insight on what databse settings could result in the best performance in terms of throughput and latency while exploiting NVM.

4. Unlike other researches done with emulated NVM devices, all presented results are produced on a server with a real commercial NVDIMM product.

5. Based on the profiling results of our final proposed scheme, we provide directions to future researches that try to exploit state-of-the-art NVM devices for better database performance.

The rest of the paper is organized as follows: Chapter 2 provides background for partial page update problem and baseline profiling results; Chapter 3 describes our optimizations; Chapter 4 describes how we actually implemented the scheme on top of MySQL 5.6.21; Chapter 5 provides evaluation results under OLTP workloads; Chapter 6 explains related work; Chapter 7 concludes the paper and Chapter 8 mentions the remaining optimization points and possible direction in future research.

Chapter 2

Background

2.1 Solving Partial Page Update Problem

Due to their essential use in many serious applications, databases should guarantee strong data consistency even in case of a sudden system crash or a power failure. One of the consistency issues that these crashes can induce is partial page update. To solve the partial page update problem, InnoDB writes page(s) to a fixed area called *doublewrite buffer* (DWB) in the storage by out-of-place update and then re-writes each page(s) to its original location (in-place update) afterwards as shown in Figure 2.1. The writes are completed forcefully using *fsync* calls to ensure that the *doublewrite* pages are written persistently before the actual data page. When the database system recovers from a system failure, it can always find consistent pages in either the database or the *doublewrite buffer* area on SSD. If the page in the database is corrupted due to partial udpate, the corresponding page from the *doublewrite buffer* area is copied to the corrupted data page to recover consistency since it is guaranteed to be flushed

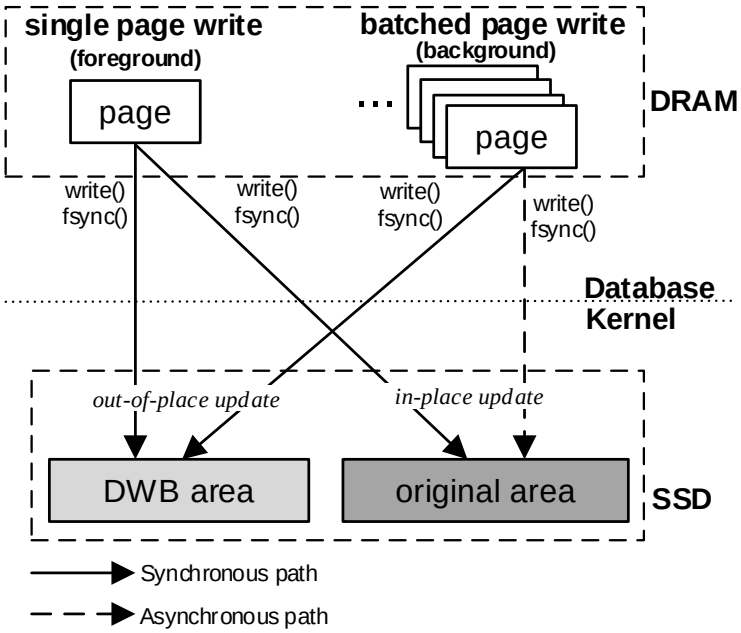

Figure 2.1 Original scheme for protecting partial page update

persistently, thus having correct page contents.

InnoDB performs flush operations based on steal policy, which might flush the pages from uncommitted transactions as well as those that are committed if necessary. There are two paths in which pages are written to the DWB in the flushing mechanism: *single* and *batched* page writes.

Batched page write is a periodically performed background flushing by the page cleaner thread based on *Least-Recently-Used* (*LRU*). It adds a page from the tail of the LRU list to the DWB in DRAM. Either periodically or when the batching list is full, the cleaner thread first writes the pages to the DWB area in file synchronously, writes the batched pages asynchronously to their original locations, and then issue an *fsync* call.

When a transaction thread is unable to find a replaceable page at the tail

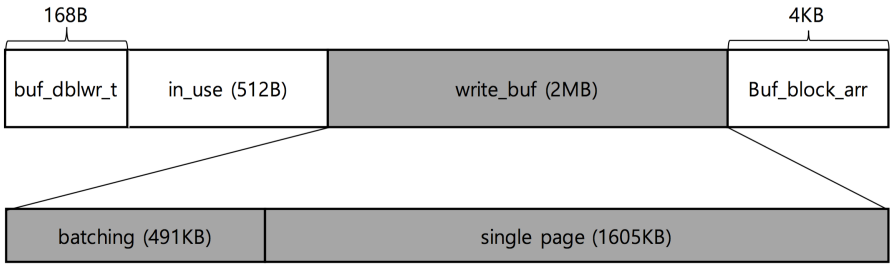

Figure 2.2 Space for *doublewrite* buffer in the original scheme

of the LRU list (i.e., when the background LRU flushing in the page cleaner thread is not fast enough to keep pace with the workload), *single page* write is called from transaction threads as foreground LRU flushing to secure a free page for page read from main storage to memory or for allocating a page for write operation. It picks up a single page from the tail of the LRU list, flushes it (if it is dirty), removes it from the list, and puts it on the free list. The transaction threads write a page to the DWB area, issue an *fsync* call, and the then write to the original location synchronously. The final step is done synchronously to immediatly acquire a free page.

On SSD-based database system, redundant writes and *fsync* calls generate significant overhead. Especially, an *fsync* call causes a long delay since it generates metadata update of the file and flushes the dirty pages in the device cache in SSD to persistent NAND chips, which usually hinder read operations.

Figure 2.2 describes the space allocated for *doublewrite* in DRAM and SSD. The whole space including *buf_dblwr_t*, *in_use* array, *write_buf*, and *buf_block_arr* are allocated in DRAM while *write_buf* also resides in the start location of system space file of InnoDB for persistence. *Buf_dbwr_t* struct, which is 168B in size, stores metadata for *doublewrite* functionality, including where to write the next page, how much space is allocated for *doublewrite*, etc. *Write_buf* is

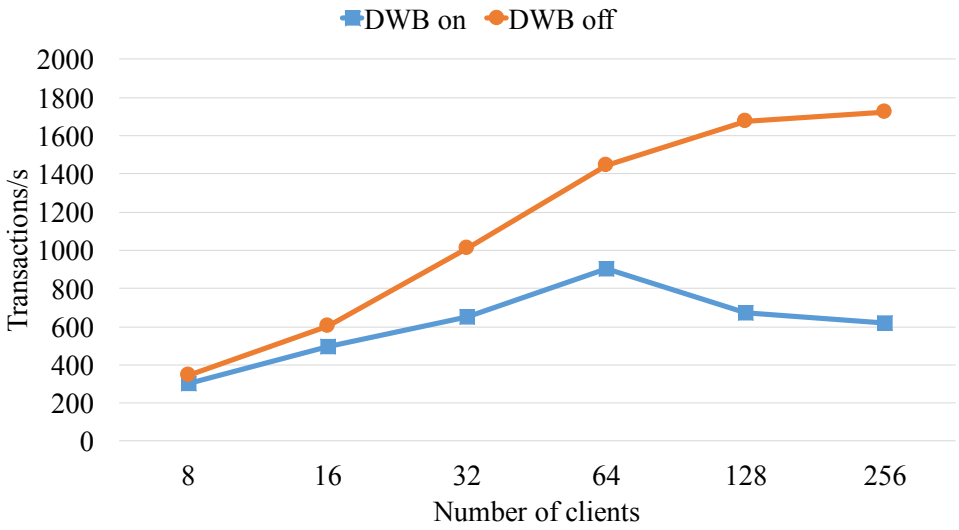

Figure 2.3 Baseline performance

the actual buffer where page contents are copied in page unit just before the data is written to the original location. Out of 2MB occupied by *write_buf*, 120 pages (491KB) are reserved for batching writes while 392 pages (1605KB) are reserved for single page writes. *In_use* is a bitmap array that are used to mark whether each page section in single page *doublewrite* region is empty or not.

2.2 Baseline Performance

To justify our reasoning that the dobublewrite scheme is a signifcant overhead in I/O intensive workloads, we have executed sysbench OLTP workloads with either *doublewrite* mode enabled or disabled. We have set up our experimental environment as described in section 4.1.

Figure 2.3 describes the experimental results. As expected, we can see that MySQL with *doublewrite* disabled gives far better throughput than the enable version; it gives 1.14X, 1.22X, 1.55X, 1.61X, 2.49X, 2.78X better throughput

Function	Total time (s)	Portion (%)
trx time	70,531	100
log_write_up_to	6,196	8.7
buf_page_get_gen	27,901	39.5
single_page_dwb_write	3,504	4.9
single_page_dwb_flush	1,264	1.79
single_page_ori_write	2,263	3.2
single_page_ori_flush	1,434	2.0
buf_flush_page	9,228	13
add_to_batch_write	567	0.8
add_to_batch_flush	177	0.25

Table 2.1 Original MySQL profiling results

than the enabled version under 8, 16, 32, 64, 128 and 256 threads respectively. We can see that the performance gap becomes more dramatic as the number of clients is increased, which means *doublewrite* is a definite obstacle to scalability due to their sequential characteristics. It can be noted that the performance with *doublewrite* disabled is our maximal performance gain that can be expected by simply placing DWB on NVM.

To further justify our reasoning, we have profiled how much time the threads spend inside the functions for *doublewrite*, of which result is described in table 2.1. Functions related to doublewrite, *single_page_dwb_write*, *single_page_dwb_flush*, *single_page_ori_write*, *single_page_ori_flush*, *add_to_batch_write*, and *add_to_batch_flush* sum up to 12.94% of the total running time of transactions. This is not a small portion but the actual *doublewrite* overhead is somewhat shadowed because single page evictions during read requests actually increase the lock contention

and waiting time inside *buf_page_get_gen*. From the baseline performance results given in this section, it could be realized that removing doublewrite overhead by exploiting NVM could result in high performance gain.

Chapter 3

Design & Implementation

3.1 Partial Page Update Protection on NVM

First, we propose a simple scheme called *NVM-DWB*, which simply allocates the *doublewrite* buffer in NVM instead of DRAM. Since NVM allows persistence, this scheme removes all file I/Os for the doublewrite buffer to SSD in both *batching* and *singe* page write paths. In order to provide guarantee that memory updates reach NVM, we perform *clflush* operations when the page(s) should be flushed and memory barriers right before and after the operation to enforce ordering on memory operations. Consequently, all doublewrite buffer operations are performed only in NVM via memory operations without calling any file system operations. In this scheme, however, there are still frequent file I/Os for writing and fsyncing pages to SSD synchronously or asynchronously by transaction threads, which is still a large I/O traffic. Moreover, read operations for evicted pages should be performed from SSD to DRAM since InnoDB only allows recovery process to read pages from doublewrite region while not

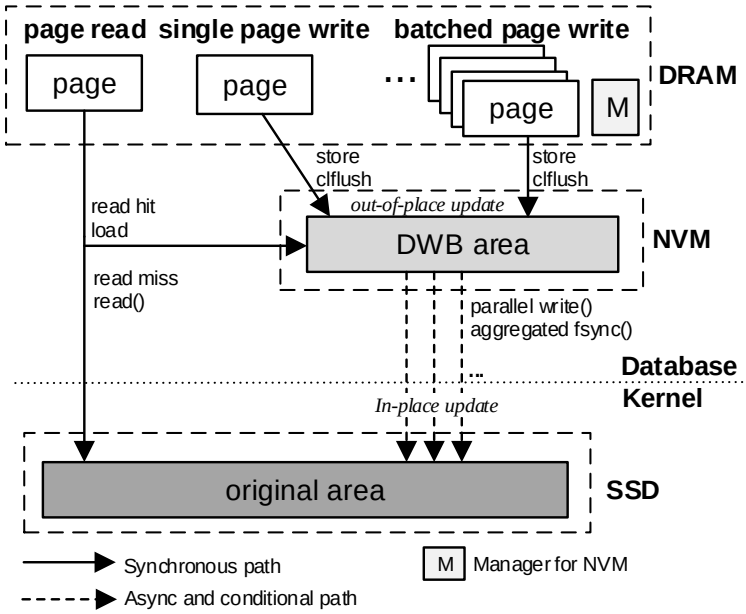

Figure 3.1 Proposed scheme

allowing user threads from reading pages in the DWB area even if the correct page contents reside there. Consequently, even though NVM is a good place for the DWB, simply placing the buffer inside NVM cannot fully exploit the performance NVM provides.

3.2 An Efficient Buffer Scheme on NVM

As stated in the previous section, pages in the DWB could be used to serve read requests for better performance; they need to go through file I/O to be served because they are already evicted from the DRAM buffer pool. Hence, our next step in optimization is to combine the protection scheme for partial page update and the buffer scheme for page reads and writes on NVM as shown in Figure 3.1. Since full page contents are written to DWB in transactions, it

could be used as a cache for read/write operations. A caveat in *NVM-DWB* is that the original DWB, which is 2 MiB in size, is too small to be used as a read/write cache. Therefore, we increase the size of the DWB to 1 GiB. It also reduces the overall I/O traffic to SSD, which improves SSD endurance due to write buffering on NVM.

To provide our scheme, we make an NVM space manager and its related operations. The NVM space manager is composed of a hash table with circular doubly-linked lists. Pages are inserted into the hash table after being hashed by their page and each node in the list maintains page id, space id, and the address of page content in NVM.

We can place and constitute NVM space manager either on NVM or DRAM area but the selection provides a trade-off. Placing it in NVM costs additional NVM space and NVM address mapping overhead. Placing it in DRAM produces additional cost of re-organizing the space manager from NVM at boot time; moreover, data structures that support fast search are more difficult to implement correctly and efficiently in NVM [15]. In this paper, we select the latter to save NVM space, reduce the managing overhead, and facilitate the implementation of our scheme.

3.2.1 Decoupling synchronous path from transaction processing

NVM-DWB scheme still generates two *write()* and *fsync()* calls synchronously in transaction processing for a single page update to its original location. In this case, the transaction threads spend additional flushing time, unlike *batched* page writes. Our proposed scheme decouples this synchronous path using the write buffer scheme on NVM. Since every page that are going to be flushed is sequentially written to DWB by out-of-place updates, we can utilized the DWB

as a write buffer. To put in another way, single page is removed and every write request is processed in batches without giving up average latency.

3.2.2 Supporting read buffer on NVM

When a transaction thread performs a read operation for a page, it first searches for the page using the NVM space manager. We specifically implement a hash table of pages written in the DWB in DRAM to efficiently support searching. Since the DWB is also used as a write buffer, there can be multiple entries with different page contents in the bucket of the hash table. However, the NVM space manager always finds the recently added page because the linked list of each bucket is implemented as a circular doubly linked list; the list can be scanned in reverse from its tail.

3.3 Re-do Logging on NVM

The experimental results from baseline performance showed that redo logging is not the most signifcant performance bottleneck in the original scheme. However, the logging could be the next performance bottleneck since it becomes one of the largest I/O overhead after *doublewrite* overhead is removed and page flushing becomes asynchronous. After supporting the optimizations mentioned in chapter, we also place log buffer on NVM and totally remove log file I/Os for further performance improvements.

3.4 Recovery and Parallel flushing in NVM

We flush the pages in DWB to disk if the area becomes full. To fully exploit the parallelism provided by flash-based SSD, we create multiple threads to process buckets in the hash table in parallel. Since order is kept between different

versions of a same page in the linked list of the hash table, so is the order that they are actually flushed to disk, thereby keeping data consistency.

Even if a crash occurs when flushing the pages in NVM, we safely recover with the existing algorithm. What differs is that our scheme reads *doublewrite* pages directly from NVM, instead of reading the file region assigned to *doublewrite* in SSD. We prepare a list of *doublewrite* pages by scanning the NVM space and selectively write to the original data location when the data is corrupt, which indicates system crash might have occured during page update.

3.5 On-the-fly page reclamation in NVM

Toward more efficient management of NVM space, we provide on-the-fly reclamation for duplicate pages in the DWB. Since we refrain from flushing pages until the DWB becomes full, there could be multiple versions of a same page in the buffer if the page is updated more than once during the interval. There is no need to flush these obsolete versions because only the last updated version is applied in an idempotent fashion. After a page content is inserted into the NVM space, the scheme checks if there is a previous version of the page by looking at the hash table with the given page id and space id. If there is, the space occupied by the previous version is marked as a hole that will be filled by the next page write. The new version could not be inserted into the space of the previous version because the writes should be done in an out-of-date fashion to prevent partial page update. On-the-fly page reclaim reduces the total number of writes by preventing possible flushing of duplicate pages, thereby increasing flash endurance of SSD as well as lengthening the period between flushing pages when the NVM space is fully filled with pages.

Chapter 4

Evaluation

In this section, we describe our experimental setup and show the evaluation results obtained from our proposed scheme by running OLTP database workloads.

4.1 Experimental setup

Our machine has two Intel Xeon CPU E5-2620 (2.10 GHz) with 6 physical cores each, which total up to 12 cores with hyperthreading 24 cores. It has 64GB of DRAM memory and 16GB of NVDIMM and equipped with a flash-based SSD (size of 256GB) (Samsung 850Pro) with SATA3 interface. We use a commercially available NVDIMM [1] as an NVM alternative since it provides persistence and DRAM-like performance [1]. We have implemented this scheme

[1]NVDIMM is a combination of DRAM and NAND flash. During normal operations, NVDIMM is working as DRAM while flash is invisible to the host. Upon power failure, NVDIMM saves all the data from DRAM to flash by using supercapacitor to make the data persistent. Since this process is transparent to other parts of the system, NVDIMM can be treated as NVM

Parameters	Values
Page size (KiB)	4
DB buffer size (GiB)	1
Flush method	O_DIRECT
Table size	100,000,000

Table 4.1 InnoDB configuration

Parameters	Values
mysql-table-engine	innodb
oltp-table-size	10,000,000
max-time	600
num-threads	1 - 256
max-requests	100,000,000

Table 4.2 Sysbench configuration

in MySQL 5.6.21 and Linux 3.14.3 with mounted EXT4 file system.

4.2 OLTP-benchmark results

In this section, we present experimental results from sysbench and tpc-c OLTP workloads. We have ran multiple macro-benchmarks for evaluating the original and proposed schemes since they might have different I/O implications such as the ratio of read or write requests in a transaction, thread scalability, or the actual internal implementation.

For sysbench, we loaded the data with table size of 100M rows, which amounts to 30GB of data. We varied the number of request clients for sysbench but kept other parameters consistent. The detailed configuration for sysbench is decsribed in table 4.2. For tpc-c, we used the warehouse size of 500, which amounts to 52 GB ofdata in total. Similar to the sysbench configuration, we varied the number of clients while keeping other parameters consistent. The

Parameters	Values
warehouse	500
connection	8 - 64
rampup time	60
measure time	600

Table 4.3 Tpc-c configuration

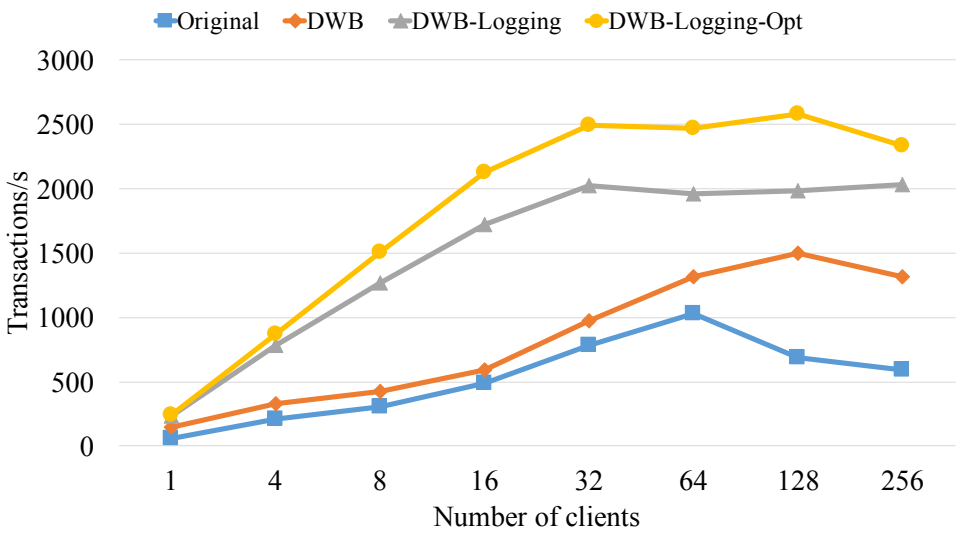

Figure 4.1 Sysbench throughput for the original and proposed schemes

detailed configuration for tpc-c is described in table 4.3. Detailed configuration for MySQL is described in table 4.1. Page size of 4 KiB is used in our experiments because they elicit the best throughput when SSD is used as a main storage device.

4.2.1 Throughput

Figure 4.1 shows the performance comparsion between the original scheme and various schemes that we have implemented. *Original* indicates the unmodified version of MySQL whereas DWB means the implementation where the

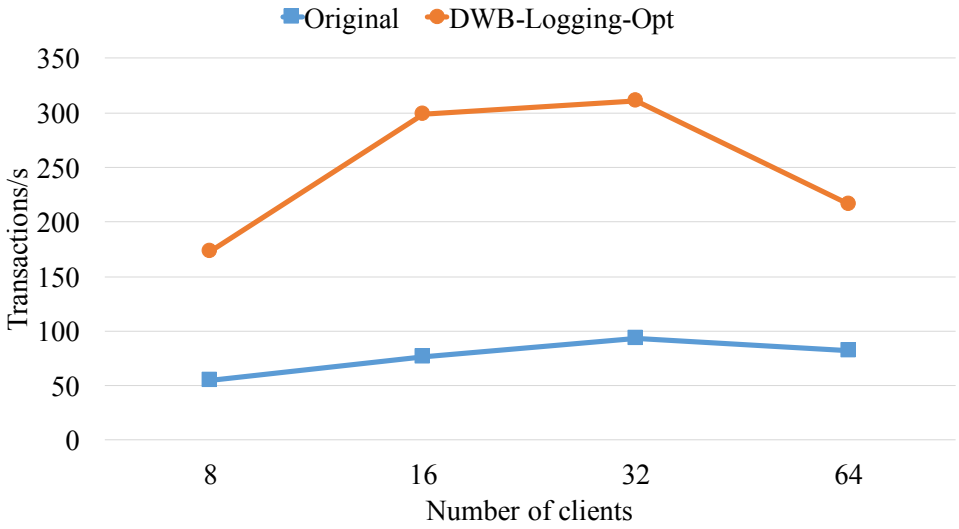

Figure 4.2 Tpc-c throughput for the original and proposed schemes

DWB are simply located on NVM. DWB-Logging version indicates the implementation where both DWB and log buffer are simply placed on NVM while DWB-Logging-opt version indicates the final implementation with all the optimizations mentioned in chapter 3.

In the original scheme, throughput scales from 16 to 64 threads but drops signifcantly after 64 threads, which could be attributed to the I/O contention induced from doublewerite and logging overhead. Comparing the original scheme and *NVM-DWB*, just removing the I/O overhead of doublewrite provides better performance at every number of clients; especially, *NVM-DWB* scales until 128 threads and show 3.76X better performance than the original scheme. We see a drastic performance improvement under even a small numbers of threads as we implement both DWB and logging on NVM. The fact that both *NVM-DWB-Logging* and *NVM-DWB-Logging-Opt* peak their performance at 32 clients and show flat performance afterwards indicate that possible other performance bot-

tlnecks have appeared since most of I/O overhead is removed. After 32 threads, *NVM-DWB-Logging* and *NVM-DWB-Logging-Opt* shows comparatively large performance gap, which indicates that our various optimizations indeed have meaningful performance benefits. In overall, the proposed scheme shows 3.94X, 4.15X, 4.94X, 4.31X, 3.19X, 2.40X, 3.76X, 3.97X better performance compared to the original scheme under 1, 4, 8, 16, 32, 64, 128, and 256 threads, respectively.

Figure 4.2 shows throughput of the original and proposed schemes using tpc-c workloads. Similar to the result from sysbench workloads, *NVM-DWB-Logging-Opt* shows 3.16X, 3.93X, 3.32X, 2.64X better throughput than the original scheme under 8, 16, 32, and 64 threads, respectively. What differs from sysbench results shown in figure 4.1 is that our proposed scheme does not scale from 32 clients to 64 clients. This could be attributed to the difference in internal implementation of the two workloads; tpc-c might contain other factors that might hinder performance at larger number of threads such as more join operations during transactions while sysbench consists of simpler queries in overall.

4.2.2 Latency

We have also compared the average response time of the original and our proposed schemes and described the results in figure 4.3. We can see that *NVM-DWB-Logging-Opt* shows far better average response time compared to the original version under every number of threads measured. *NVM-DWB-Logging-Opt* shows average latency below 50 ms until 128 threads and 110 ms at 256 threads while *Original* shows average latency greater than 140 ms after 32 threads, which are unacceptable in user point of view. In conclusion, it shows 4.98X, 4.30X, 3.67X, 3.97X, 5.54X, 3.77X, 3.76X better average response time

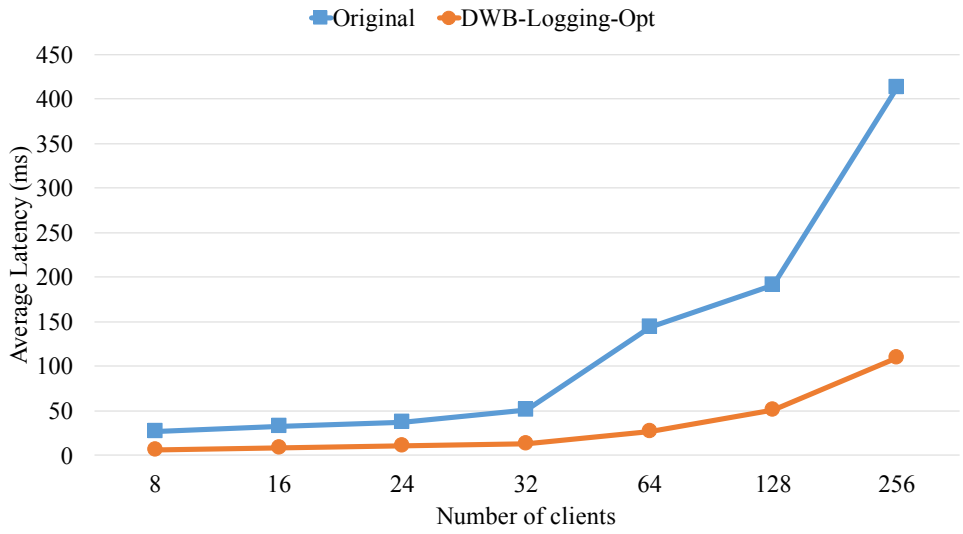

Figure 4.3 Average response time of the original and proposed schemes

under 8, 16, 24, 32, 64, 128, and 256 threads, respectively.

Table 4.4 shows the profiled results from our final proposed scheme. Compared to the original profiled results shown in table 2.1, the overhead of *single* page write paths and *buf_flush_page* are totally removed since every page flushing is done asynchronously. In addition, logging overhead is dramatically reduced from 8.7 % to 0.018 % since only memory copy operation to NVM remains after log file I/Os are removed. This result indicate that now the performance bottleneck has shifted to other portion of the program after I/O overhead is drastically reduced. More detailed profiling indicates that the reader-writer lock contention coming from searching and modifying index trees now comprises major portion of the running time. Especially, readers are spending much more time sleeping while writers are holding the lock in exclusive mode, of which overhead could be signifcantly relieved by implementing some variants of RCU that might allow non-blocking read.

Function	Total time (s)	Portion (%)
trx time	76,651	100
log_write_up_to	14.39	0.018
buf_page_get_gen	30,195	39.4
single_page_dwb_write	0	0
single_page_dwb_flush	0	0
single_page_ori_write	0	0
single_page_ori_flush	0	0
buf_flush_page	436	0.57
add_to_batch_write	307	0.4
add_to_batch_flush	177	0.25

Table 4.4 Proposed MySQL InnoDB profiling results

Function	Count	Total amount (GB)
buf_dblwr_add_to_batch	7,383,654	28.2 GiB
fil_io (write)	5,727,788	21.8 GiB
log_write_up_to	8,940,275	4.2GB

Table 4.5 Call counts for functions related to write I/O

4.2.3 NAND Flash Endurance

In this section, we explore how much endurance can be improved due our proposed scheme. There are mainly two parts in our implementation that improves flash endurance: on-the-fly page reclamation and logging on NVM. On-the-fly page reclamation reduces the number of page flushed by skipping older versions of duplicate pages and logging on NVM totally removes redo log writes by placing it on NVM and removing log files.

The number of pages that are actually flushed to SSD during page flushing in the proposed scheme could be calulcated from the information provided in Table 4.5. Since all page writes are done in batches, the number of times the function buf_dblwr_add_to_batch is called matches the number of toal page

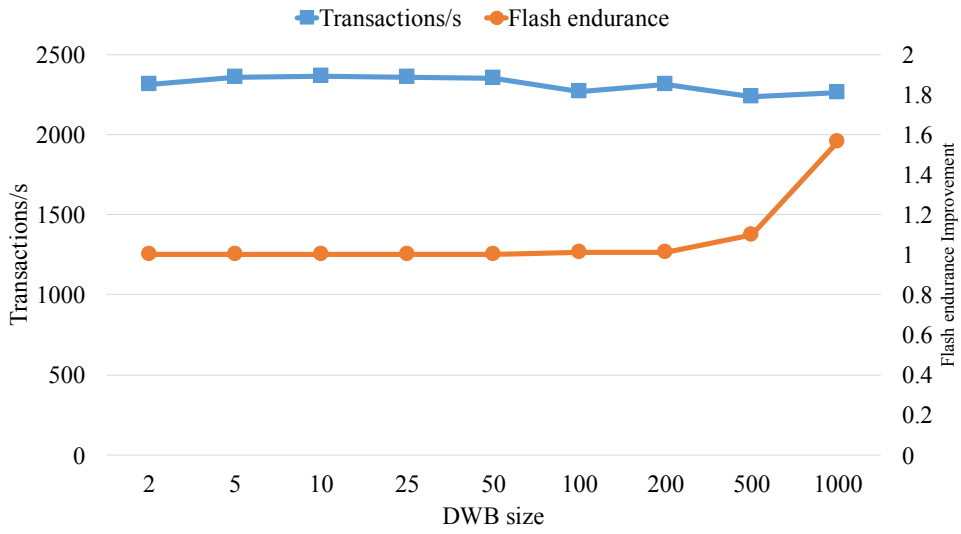

Figure 4.4 Throughput and flash endurnace over varying DWB size

write requests. *Buf_dblwr_add_to_batch* is called 7,383,654 times during 600s of sysbench workload and the actually write file I/O are called 5,727,788, which are 28.2 GiB and 21.8 GiB, considering the page size is 4 KB in our settings. Therefore, we saved 6.4 GiB of data from being actually written to storage during flushing. We can see that *log_write_up_to* was called 8,940,275 times for 10 minute run of sysnbench workload, which amounts to 4.2 GiB of log data since each log block occupies 512 B. Thus, we prevented 4.2 GiB of log data from being written to SSD by exploiting NVM to remove log file and related file I/Os. In conclusion, including the original 28.2 GiB of doublewrite pages that are prevented from being written to SSD, our proposed scheme substantially improves flash endurance by 2.78X from 60.6 GiB to 21.8 GiB.

4.2.4 Varying DWB size

Thus far, all experiments have been done with a fixed size of DWB area using 1 GiB of NVM. In this section, we explore how varying the size of NVM space

for DWB area can affect throughtput and flash endurance. Figure 4.4 shows the throughput and flash endurance improvement over different sizes of DWB area on NVM. The workload is same as described in Table 4.2 but the number of threads is fixed at 128 threads. Flash endurance improvement is a ratio of write I/O amount with 2 MiB of DWB area to the amount with the specific DWB size; it is 1 when the default size for DWB (2 MiB) is used for NVM. It gradually increases to 1.097 until 500 MiB of DWB area and suddenly reaches 1.564 with 1 GiB. The value of 1.564 means that more than 36% of writes were saved due to on-the-fly page reclamation, which does not flush old versions of duplicate pages in the DWB area. It can be noted that flash endurance is greatly improved as DWB size is increased from 500 MiB to 1 GiB, which indicates that on-the-fly page reclamation becomes drastically effective with 1GiB of DWB area.

However, throughput shows a rather flat curve as DWB size increases. This observation indicates that decoupling synchronous path from transaction processing and NVM logging are main factors in increasing the performance of our optimizations while building hash tables for pages in DWB area mainly contributes to improving flash endurance by allowing removing duplicate pages from the area.

Chapter 5

Related Work

There have been researches on how to take advantage of NVM for improving database performance as NVM technologies have evolved and several companies have started to sell commercially ready products. One stem of these researches focuses on exploiting NVM for better logging performance, which was considered the performance bottleneck for databases.

PCMLogging exploits the features of NVM to integrate logging and data caching. However, this approach does not support the transactional atomicity and durability [4] and its tuple-based updating by exploiting byte-addressable feature of NVM does not maintain all page contents on PCM. In addition, differently from PCMLogging, we deal with the partial page update problem from a real implementation.

With the help of NVM, Wang et al. [14] achieves distributed logging, which was considered notoriously hard under single-node systems in the DRAM era. They achieve 3X speedup with distributed logging on multicore and multi-socket hardware and passive group commit compared to the centralized logging

.

Similar to our direction, some studies have tried to relieve the performance bottleneck induced by the protection scheme for partial page upate such as InnoDB *doublewrite*. Kang et al. [7] proposes utilizing the doublewrite buffer in SSD as read cache in HDD-based database system by increasing the size of the *doublewrite* buffer. Their approach is similar to ours in that page contents in doublewrite buffer are used to serve read requests and they achieve around 50% of performance improvement by serving 40% of read requests to SSD. However, there still remains file I/Os for DWB since the system utilizes flash-based SSD for doublewrite buffer while maintaining hard disk for main storage. Their system still suffers from I/O overhead and endurance problem of SSD. We basically claim that NVM is a good placement for doublewrite buffer and perform several optimizations using features of the NVM.

DuraSSD [6], which is written by the same author as [7], totally removes the doublewrite buffer by exploiting internal durable cache and modifying firmware inside SSD. However, this requires additional modifications to hardware as opposed to our scheme.

Chapter 6

Conclusion

In this paper, we have presented a study on leveraging NVM to support efficient protection of partial page update. Our scheme is to integrate the technique protecting partial page update and page buffer scheme by exploiting features of NVM such as persistence, fast access, and byte-addressability. We decouple synchronous flushing path from transaction processing, implement read/write buffer on NVM in addition to buffer pool, and reclaim doublewrite space for obsolete versions of pages for better database performance. Moreover, we place redo-logging in InnoDB on NVM and completely remove redo log file to further reduce I/O overhead. The experimental results show that our final scheme with all these optimizations shows 4X throughput, and 1.56X flash endurance compared to the existing scheme in InnoDB.

Chapter 7

Future Work

In our scheme, we have removed most of I/O overheads coming from flushing pages and redo logging except when the flushing is done in batches. Our profiling results after the final implementation show that now the bottleneck has moved from I/Os to lock contention. Table 4.4 clearly shows how much time threads are spending due to lock contention. Since rw_lock_s_lock and rw_lock_x_lock, which are functions for acquiring locks in shared and exclusive mode respectively, comprise a huge portion of the total time, additional idea for removing the lock contention, such as implementing more efficient lock manager and lock-free data structures, might be beneficial in increasing the database throughput to further exploit NVM performance.

Chapter 8

Bibliography

[1] AGIGATECH. Agiga tech - agigaram. `http://www.agigatech.com/agigaram.php`, 2015.

[2] R. Fang, H. I. Hsiao, B. He, C. Mohan, and Y. Wang. High performance database logging using storage class memory. In *2011 IEEE 27th International Conference on Data Engineering*, pages 1221–1231, April 2011.

[3] Hemant Gaidhani. Understanding ssd endurance. `https://itblog.sandisk.com/ssd-endurance-speeds-feeds-needs/`, 2015.

[4] Shen Gao, Jianliang Xu, Bingsheng He, Byron Choi, and Haibo Hu. Pcmlogging: Reducing transaction logging overhead with pcm. In *Proceedings of the 20th ACM International Conference on Information and Knowledge Management*, CIKM '11, pages 2401–2404, New York, NY, USA, 2011. ACM.

[5] Jian Huang, Karsten Schwan, and Moinuddin K. Qureshi. Nvram-aware logging in transaction systems. *Proc. VLDB Endow.*, 8(4):389–400, December 2014.

[6] Woon-Hak Kang, Sang-Won Lee, Bongki Moon, Yang-Suk Kee, and Moonwook Oh. Durable write cache in flash memory ssd for relational and nosql databases. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 529–540, New York, NY, USA, 2014. ACM.

[7] Woon-Hak Kang, Gi-Tae Yun, Sang-Phil Lim, Dong-In Shin, Yang-Hun Park, Sang-Won Lee, and Bongki Moon. Innodb doublewrite buffer as read cache using ssds. In *In 10th USENIX Conference on File and Storage Technologies*, 2012.

[8] Raymond A. Lorie. Physical integrity in a large segmented database. *ACM Trans. Database Syst.*, 2(1):91–104, March 1977.

[9] C. Mohan. Disk read-write optimizations and data integrity in transaction systems using write-ahead logging. In *Proceedings of the Eleventh International Conference on Data Engineering*, ICDE '95, pages 324–331, Washington, DC, USA, 1995. IEEE Computer Society.

[10] C. Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter Schwarz. Aries: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Trans. Database Syst.*, 17(1):94–162, March 1992.

[11] Vidyabhushan Mohan, Taniya Siddiqua, Sudhanva Gurumurthi, and Mircea R. Stan. How i learned to stop worrying and love flash endurance.

In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Storage and File Systems*, HotStorage'10, pages 3–3, Berkeley, CA, USA, 2010. USENIX Association.

[12] PostgreSQL. Postgresql - documentation: 9.5 - write ahead log. `https://www.postgresql.org/docs/current/static/runtime-config-wal.html`, 2015.

[13] Baron Schwartz, Peter Zaitsev, and Vadim Tkachenko. *High Performance MySQL, 3rd Edition*. O'Reilly, 2012.

[14] Tianzheng Wang and Ryan Johnson. Scalable logging through emerging non-volatile memory. *Proc. VLDB Endow.*, 7(10):865–876, June 2014.

[15] Jian Xu and Steven Swanson. Nova: A log-structured file system for hybrid volatile/non-volatile main memories. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 323–338, Santa Clara, CA, February 2016. USENIX Association.

초록

이 논문에서는 플래시 SSD 기반의 데이터베이스에서 부분 페이지 갱신 방지 와 버퍼 기법에 비휘발성 메모리를 도입함으로써 성능을 높이는 효율적인 방법을 제시한다. InnoDB의 doublewrite 또는 PostgreSQL의 full-page write처럼 데이터베이스의 부분 페이지 갱신 방지를 위합 기법들과 페이지 버퍼 기법을 비휘발성 메모리를 사용하여 통합함으로써 읽기와 쓰기에서 SSD I/O와 그에 수반되는 lock contention을 현저히 줄이는 것이 목표이다. 이에 더하여 InnoDB의 redo log buffer 자체를 비휘발성 메모리에 올림으로써 모든 I/O를 비동기적으로 발생시켜 더욱 더 높은 성능 향상을 꽤하였다. 우리는 이러한 적화 기법을 NVDIMM이 장착된 서버의 MySQL InnoDB에 올림으로써 기존과 비교하여 트랜잭션 성능을 네 배 이상, 플래시 내구성을 3배 이상 끌어올리는데 성공하였다.

주요어: 데이터베이스, 부분 페이지 갱신, 페이지 버퍼 기법, 비휘발성 메모리, byte-addressable, I/O, redo logging

학번: 2014-22679