# Efficient Write Ahead Logging with Parallel Write

Komei Kamiya
University of Tsukuba, JST CREST
Email: kamiya@hpcs.cs.tsukuba.ac.jp

Hideyuki Kawashima
University of Tsukuba, JST CREST
Email: kawasima@cs.tsukuba.ac.jp

Osamu Tatebe
University of Tsukuba, JST CREST
Email: tatebe@cs.tsukuba.ac.jp

***Abstract*—The collisions incurred by log writes by multiple worker threads for the write ahead logging (WAL) protocol degrades the performance of transaction processing. In this paper, we propose a new WAL protocol, P-WAL. To deal with performance degradation problem incurred by I/O access concentration with sequential access and thread blocking with WAL buffer access, P-WAL writes log records in parallel. In the scheme, workers do not share neither the WAL buffer nor the WAL file. Each worker thread writes log records to its corresponding storage space in parallel. This incurs random accesses for the storage device. It degrades performance degradation on hard disk, but it does not matter for the ioDrive because ioDrive involves the parallel access mechanism. A bottleneck remains despite of P-WAL scheme. It is the access to the log sequence number. We employ fetch_and_add operation to alleviate the problem. We implement a prototype in-memory transaction processing system, and evaluate the performance through experiments. P-WAL achieves 425,531 tps in our micro benchmark. It is around 2.42 times higher throughput compared with that of conventional method.**

## I. Introduction

The volume of transaction processing is becoming immense in these days. NYSE provides less than 10 micro second response time [6]. To accelerate transaction processing, a variety of works have been studied. Transaction processing needs to provide ACID natures, and it is known that on the failure of a database system, the state of the system must be consistently recovered. This feature is achieved by the atomicity and the durability.

To provide atomicity and durability, a transaction system usually executes write ahead logging (WAL) or journaling. WAL writes a log record that includes update information to storage before modifying database objects in the persistent storage. WAL provides recover-ability to a database system while it degrades performance of transaction processing because it requires a huge volume of storage accesses. Current WAL algorithms assume that its underlying storage is a hard disk drive (HDD). Therefore, the current WAL algorithm is designed to minimize the expensive IO cost for the HDD, and it tries to avoid random accesses. It first gathers multiple transaction logs in the memory, and then it transfers them in a lump to a single log file on the storage. This scheme is adopted by usual database systems including PostgreSQL.

On the other hand, recently we see that a new type of storage devices is emerging. One of such devices is the ioDrive that provides different performance characteristics comparing with HDD. Therefore, it is unclear whether the current WAL protocol can fully leverage the feature of ioDrive, and whether it achieves the maximum performance with the ioDrive.

To figure out the riddle, we first clarify the performance feature of ioDrive, and then propose a new WAL protocol, P-WAL that exploits the feature of the ioDrive. In P-WAL, a worker thread occupies a dedicated WAL buffer to eliminate locks to the WAL buffer, and it also occupies a dedicated WAL storage segment. This scheme generates parallel random write operations frequently. Such workloads should be avoided in HDD, while it is suitable in ioDrive. Although locks required for the buffer accesses are totally eliminated by the proposed design, collisions happen when incrementing the log sequence number (LSN). To alleviate this problem, we use fetch_and_add instruction. In addition, we propose an efficient redo phase protocol that does not use the expensive sorting. We evaluate the performance of P-WAL with both the micro benchmark and the TPC-C New Order benchmark, and show the excellent performance of P-WAL.

As for related work, some works accelerate the WAL keeping the ARIES scheme [5]. They are Aether [2], Deuteronomy [4], and distributed logging [8]. Aether and Deutronomy do not provide the parallel storage access method while P-WAL does it. Distributed logging and FOEDUS [3] show high scalabilities and high performances. However, they assume that their underlying storages as NVRAM, which requires the re-design of transaction processing system. On the other hand, P-WAL is based on ARIES protocol, and therefore it can be easily plug-gable to conventional database systems such as PostgreSQL and Oracle.

The rest of this paper is organized as follows. Section 2 describes write ahead logging. Section 3 describes storage system. Section 4 describes proposed method. Section 5 describes design of P-WAL. Section 6 describes evaluation. Section 7 describes related work. Section 8 describes conclusions.

## II. Write Ahead Logging

Write ahead logging (WAL) is a technique that provides atomicity and durability for a transaction system. WAL writes log records into a storage device before updating database objects on the storage device (Figure 1). First, a transaction generates log records based on operations in a transaction, and the records are inserted into a shared buffer on the physical memory. Second, the stored log records in the buffer are transferred to storage in a lump, which is referred to as the group commit. On the recovery after the system failure, transaction manager judges the success of each transaction by

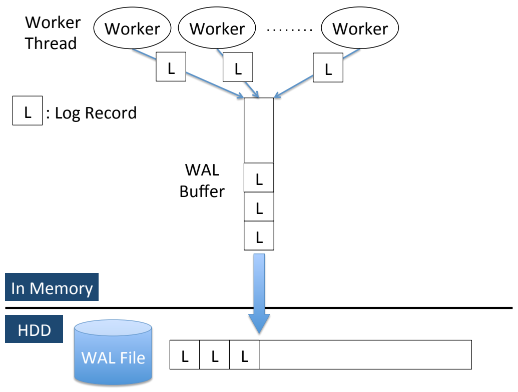

Fig. 1. WAL Architecture

**Algorithm 1** log_insert_normal(log)

```
1: WALbuffer.lock()
2: $LSN \leftarrow$ WALbuffer.insert(log)
3: if log.Type == 'COMMIT' then
4:     #Increment counter for commit log
5:     WALbuffer.ncommit $\leftarrow$ WALbuffer.ncommit $+1$
6:     if WALbuffer.ncommit == NGROUP or WALbuffer.full () then
7:         WALbuffer.flush()
8:     end if
9: end if
10: WALbuffer.unlock()
11: return $LSN$
```

checking the existence of its commit log record. Log records for committed transactions are applied to database, and updates by uncommitted transactions are deleted.

WAL provides atomicity and durability, while it degrades performance of transaction system for its overhead that includes storage accesses and locks for sharing the buffer. A study points out that the ratio of CPU instructions for logging is around 11%, while the one for useful work is 6.8 % [1]. Therefore, the acceleration of WAL is important to achieve high efficiency in a transaction system.

### A. Insertion of Log Records

We show a procedure that a transaction inserts log records to WAL buffer in Algorithm 1. First, a worker locks a WAL buffer, and then it inserts a log record to the WAL buffer. At this time, the log sequence number (LSN) is provided for the log record. LSN identifies a log record and it is necessary for recovery in the ARIES scheme. After the insertion, the worker unlocks the WAL buffer. If the number of log records in the WAL buffer exceeds threshold of group commit, then all the log records in the buffer are transferred to a WAL file. Bothe the insertion of a log record to a WAL buffer and transfer of log records to a WAL file need locks. Therefore, only one thread can run during the operations, which blocks other threads run. It clearly causes performance degradation.

TABLE I
ENVIRONMENT

| | |
|---|---|
| CPU | Intel(R) Xeon(R) CPU E5-2665 $\times$ 2 |
| # Cores | 8 $\times$ 2 |
| Memory | 64GB |
| ioDrive | SLC, 160GB, VRG5T VSL ver. 3.3.3, Low-Level Formatting |

## III. Storage Device

A conventional DBMS assumes its underlying device as a HDD, and its WAL protocol is designed based on a HDD. On the other hand, we see a variety of new devices such as SSD, NVRAM and ioDrive in these dayes. Since their performance features are different from that of HDD, it is not clear whether the conventional WAL design performs efficiently on such new devices. In this section, we describe performance features of ioDrive as the preparation of our proposal.

### A. Storage Device

HDD is a typical storage device. To access data in HDD, a magnetic head mechanically interacts with platters inside HDD. To find requested data, the head moves on the track on data, and the head waits for data during platter rotation. For the mechanical reason, IO latency of HDD is quite long. It is obvious that random access degrades IO performance dramatically on HDD.

ioDrive is a type of flash storage devices, and is connected via PCI Express. Most of storage devices are connected via south bridge such as SATA, but PCI Express is located on north bridge and it is near CPU. Recent Intel CPU (after Nehalem) involves PCI Express controller inside, and therefore CPU is able to access ioDrive directly. The ioDrive provides virtual storage layer (VSL) that maps logical space and physical space of the flash device. The mapping is done by CPU on the host side, which provides low latency. For example, when the number of threads that write to ioDrive is 1, performance of random write is the same as that of sequential write.

### B. Basic Performance of ioDrive

We show basic performance of the ioDrive in this section. We conduct sequential write operations and random write operations. For the result of random cases, the number of threads are set to 1 and 16. We show experimental environment in Table I. The result is illustrated in Figure 2. It is shown that as the size of block increases, throughput increases. Interestingly, random write (16 threads) shows the best performance among the three cases. This feature of ioDrive is totally different from those of HDD.

## IV. P-WAL: Efficient WAL exploiting High Performance Random Write

### A. Problem on Conventional WAL

There are two problems on the conventional WAL. First one is a locking procedure that is necessary to insert log records to WAL buffer. Second one is I/O accesses to a WAL file on

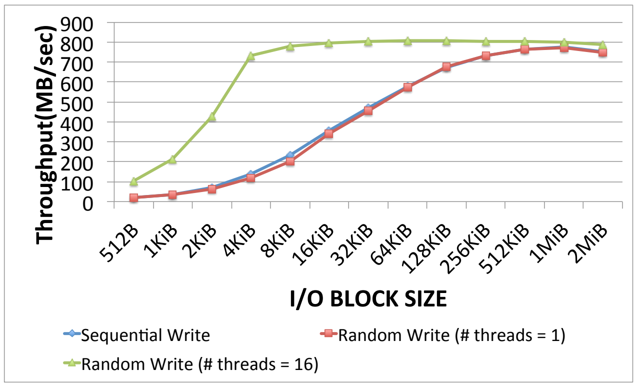

Fig. 2. Performance of ioDrive

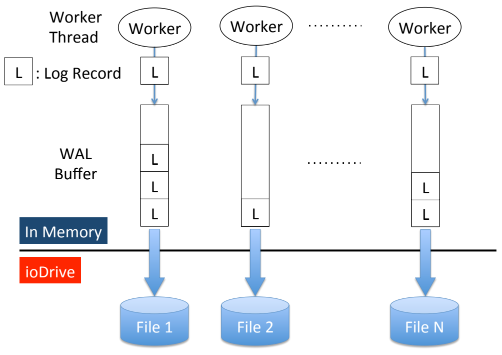

Fig. 3. P-WAL Architecture

a storage device. The HDD has been a major storage device for a DBMS, and IO cost has been considered as the most expensive operation on WAL. It can be possible to alleviate the expensive cost by exploiting the feature of ioDrive that we showed in the above.

## B. P-WAL

We propose a new WAL protocol, P-WAL here. P-WAL exploits high performance random write involved in ioDrive. In P-WAL, a worker thread exclusively uses its own WAL buffer and its own WAL file. The entire picture of P-WAL is shown in Figure 3. Comparing with the conventional architecture shown in Figure 1, P-WAL divides a WAL buffer and a WAL file which are shared in conventional one as shown in Figure 1.

*1) Division of WAL buffer:* In conventional architecture, the number of WAL buffers is one. Therefore, the insertion of a log record incurs collisions. To alleviate the collision, P-WAL provides an exclusive WAL buffer for each worker thread. The insertion to a WAL buffer is dominated by a single corresponding worker threads, and therefore each worker does not need to issue lock operations at all to insert log records to a WAL buffer.

**Algorithm 2** log_insert_pwal(log,buffer_id)

```
1: $LSN \leftarrow$ WALbuffer[buffer_id].insert(log)
2: if log.Type == 'COMMIT' then
3:     WALbuffer[buffer_id].ncommit $\leftarrow$ WALbuffer[buffer_id].ncommit $+1$
4:     if WALbuffer[buffer_id].ncommit == NGROUP or WALbuffer[buffer_id].full() then
5:         WALbuffer[buffer_id].flush()
6:     end if
7: end if
8: return $LSN$
```

*2) Insertion of Log Records:* We show an algorithm that inserts log records to WAL buffer on P-WAL. It is referred to as log_insert_pwal and shown in Algorithm 2. The difference between log_insert_pwal and conventional log_insert_normal is that log_insert_pwal does not need any locks and it inserts log records to a WAL buffer in parallel. The new argument buffer_id identifies a WAL buffer for a worker thread to insert log records. When the number of log records is more than the threshold of group commit, then log records in a WAL buffer are transferred to a WAL file in a lump.

## C. Recovery Protocol

P-WAL uses multiple WAL files. Therefore, conventional redo phase in ARIES protocol cannot be applied directly. A naive solution is sorting log records. Because it takes $O(NlogN)$ in average, it should be avoided. We propose a more efficient method in the below.

*1) Order of Log Records:* In our proposed protocol, the order of log records among WAL files are not known, but the order of the log records in a WAL file is known in prior. Therefore, we propose to solve the order of log files during WAL files by using log sequence number (LSN). An LSN is a monotonically increasing logical log ID.

During the recovery phase, log records are processed in the order of LSN. The processing is the same as the merge phase in the merge sort. It should be noticed that an LSN in P-WAL does not show the location in a log file, and thus a worker thread adds both the WAL file id and the offset in a file as appendix information to log records.

The recovery phase is shown in Figure 4. First, the head records in WAL files (LSN=11, LSN=12, LSN=14) are picked up. Among them, the smallest LSN is 11, and thus it is processed. The algorithm of recovery is shown in Algorithm 3.

*2) Efficient LSN Access:* An LSN of a log record is provided before inserting the record to WAL buffer by accessing a global variable global_LSN. Collision occurs for the access because multiple worker threads access this variable concurrently. A naive way for the concurrent accesses is the use of locks, which is notorious for its inefficiency. Therefore we adopt fetch_and_add instruction which is more efficient than CAS (Compare-And-Swap) in our environment.

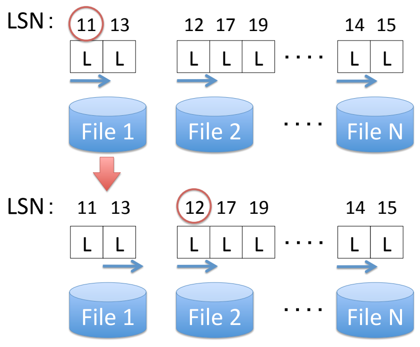

Fig. 4. Recovery by Merge

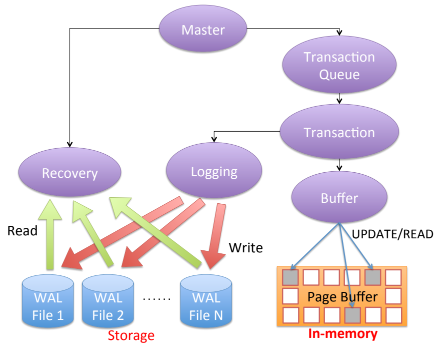

Fig. 5. System Architecture

**Algorithm 3** redo()

```
1: while Unprocessed log exist do
2:   min_LSN ← $\infty$
3:   selected ← -1
4:   for $i = 1$ to $N$ do
5:     log ← WAL_file[$i$].head()
6:     if log.LSN < min_LSN then
7:       min_LSN ← log.LSN
8:       selected ← $i$
9:     end if
10:  end for
11:  if selected ≠ -1 then
12:    log ← WAL_file[selected].next()
13:    process(log)
14:  end if
15: end while
```

## V. Design and Implementation

This section describes the design and implementation of the prototype transaction manager (TM). The TM is constituted of multiple modules: master, buffer, transaction queue, transaction, logging, and recovery. The construction of modules is shown in Figure 5. Ellipses show modules and rectangles show WAL files. Green lines show read operations from WAL files and red lines show write operations to WAL files. Black lines show communications between modules, and blue lines show updates on a page buffer.

### A. Modules

The master module checks the state of the system on the previous termination. If the termination was not legal, it conducts recovery procedure. After the recovery, it starts transaction processing. The state of previous termination is recorded in last_exit field in the master record. On the legal terminations, this field is set to "true". Thus "false" means illegal termination.

The transaction queue module manages transactions to be processed in a queue. This module provides creaters and workers. Creaters append transactions to the queue, while workers execute transaction processing. A worker has a corresponding own queue. This module sets all of transactions to the queues at the initialization phase. Therefore, collisions between creaters and workers do not occur during the transaction processing in our experiment.

The buffer module manages database pages. Because the database is designed as an in-memory database system, this module does not need to access storage for page replacement. Its concurrency control protocol is strict 2 phase lock (S2PL). Its lock granularity is page level, and pthread library is used for the lock implementation. According to ARIES protocol, updates on pages can appends to entries in the dirty page table. It is implemented by a chained hash table. A page is constituted of a page header and a page body. A page header has PageID and PageLSN. The pageID identifies a specific page. The pageLSN is an LSN of the log record that updated the page for the last time. A page body contains only an integer object in our implementation.

The Logging module provides functions for the logging. The details are described in the above. A log record contains an LSN, a previously updated value of the object, an after updated value of the object, an ID of WAL file, a file offset in the WAL file, and an LSN of undo log record. This implementation is based on the redo-undo recovery scheme based on the original ARIES protocol.

Recovery module has analysis, redo, and undo passes according to ARIES protocol. For the redo phase, we design the merge protocol as described in the above. The protocol first reads a block of records from each WAL file, and conducts the redo phase by comparing LSNs of the head of log records in the blocks. Our system does not implement checkpoint, and therefore analysis pass start from the beginning of log records.

### B. WAL File

We implement the WAL files as a raw device file on the ioDrive. Device files of ioDrive appear under the "/dev/" directory in the Linux file system. For write operations to the device files, the range of write segment should be managed by applications, which is different from usual files provided by file systems. The usage of the device file expects more efficient storage access performance compared with the file system because it bypasses the buffer cache and the meta data updates inside the file system.

To treat a device file as multiple WAL files, we provide an offset as the borderline of segments. It means that the borderline expresses the head of a WAL file. For example, if the offset is set to 10 GB, then first segment starts from the address 0, and the second segment starts from the address 10 GB. It is not allowed to exceed the borderline, but we do not need to care it because ioDrive provides the huge logical space compared with the physical space.

When writing a log record to a device file, both a header and a body should be transferred to a device file. One important issue is that headers should be written after bodies. It is because in the redo procedure, the recovery module first reads the header and then reads its body next. Please assume that the order of writes is header first, and body part is not written because of a sudden system failure. Then, redo reads only header and reads inappropriate storage area as if it is meaningful log body, which would destroy the database. Based on our scheme, on the other and, the write operation of a log header signals commit. Therefore the above inconsistency problem does not happen.

## VI. Evaluation

We evaluate the proposed method in this Section. We show the details of environment for the experiment in the Table I. The environment is the same as the one for the basic performance evaluation of ioDrive described in Section 3.

### A. *Micro Benchmark*

To understand the performance of P-WAL, we implement a simple micro benchmark. In the benchmark, we use three types of workloads by changing the ratio of update operations. In the first workload "UPDATE-1", a transaction issues only one update operation, In the second workload "UPDATE-5&READ-5", a transaction issues 5 updates and 5 reads. In the third workload "UPDATE-10", a transaction issues 10 updates.

The size of a log record is 512 bytes. By using the group commit, log records for 16 transactions are transferred to a storage device in a lump. Each operation randomly chooses a page from 65536 pages in the memory and executes a task (i.e. read or update). Read locks and write locks are conducted for read operations and update operations respectively.

UPDATE-1 generates 3 log records (BEGIN, UPDATE, END), UPDATE-5&READ-5 generates 7 log records (BEGIN, 5 UPDATEs, END), and UPDATE-10 generates 12 log records (BEGIN, 10 UPDATEs, END).

The result of experiments is shown in Figure 6. In this experiment, we compare the performance of P-WAL and conventional WAL. We show the result of experiment when the number of threads is 16 in Figure 7. When the number of threads is 16 on UPDATE-1, P-WAL showed 425,531 transactions/sec, which is 2.42 times higher throughput compared with that of conventional WAL. On UPDATE-5&READ-5 workload, the improvement is 1.75, and the one on UPDATE-10 is 1.91.

The result shows that as the ratio of update operations increases, the performance improvement degrades. The reason

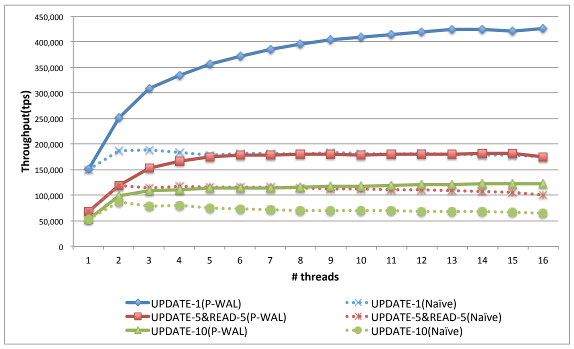

Fig. 6. 3 Workloads

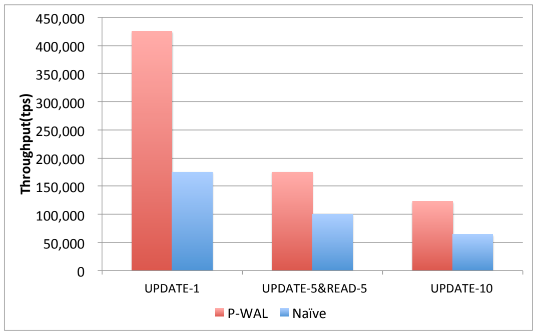

Fig. 7. Result on 16 threads

is not yet clarified, but the reasons can be because of increasing number of collisions for LSN accesses. Though we adopt fetch_and_add operation for the LSN accesses, it may not be enough to fully alleviate the collisions.

### B. *TPC-C Benchmark*

In addition with the micro benchmark, we also conduct a general benchmark, the TPC-C New-Order transactions. In the evaluation, we set scale factor to 1, which means there is only one warehouse. We compare conventional WAL and P-WAL using it. New-Order transaction simulates the order processing of a commerce. To store a page used in TPC-C, the size of log record is set to 1024 bytes. The parameter for the group commit is the same as that of the micro benchmark, which is for 16 transactions. The number of types of commerce in a single order task (ol_cnt) is randomly generated by benchmark applications. 1% of transactions in the workloads occur rollback. The contents of normal transactions are as follows. SELECT: $2 + 2 \times ol\_cnt$, UPDATE: $1 + ol\_cnt$, INSERT: $2 + ol\_cnt$. Algorithm 4 shows the algorithm of New Order transactions.

The result of experiment is shown in Figure 8. When the number of threads is 1 to 6, performances of P-WAL and conventional WAL is almost the same. When the number is 16, then P-WAL shows 1.18 times higher throughput compared with the conventional WAL.

The performance improvement in this benchmark is smaller compared with the one for the micro benchmark. The reason

**Algorithm 4** New-Order Transaction

```
BEGIN;
SELECT FROM Warehouse;
SELECT FROM District;
UPDATE District;
INSERT INTO Order;
INSERT INTO NewOrder;

LOOP
SELECT FROM Item;
SELECT FROM Stock;
UPDATE Stock;
INSERT INTO OrderLine;
END LOOP

COMMIT;
```

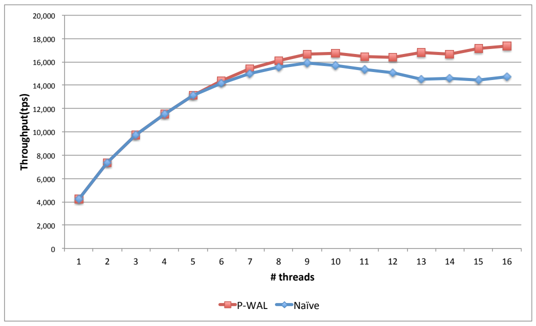

Fig. 8. Result of TPC-C (New-Order)

may be because of number of warehouses. Since there is only one warehouse, collision happen frequently in this case. The collision can be reduced by increasing the number of warehouses, but we left the work in the future.

## VII. Related Work

Efficient WAL techniques can be classified into two categories. First category is based on ARIES scheme and it uses the global log sequence number to provide an order to log records. Second category does not use the global log sequence number and re-designs the WAL.

Works in the first category includes Aether [2], Deuteronomy [4], and GSN [8]. Aether alleviates collisions by early lock release and asynchronous group commit. Deuteronomy isolates transaction components and data components and provides a flexible system design. These works also use LSN and they do not exploit parallel IO. They do not provide parallel log buffers proposed in this paper. GSN uses a new type of global sequence number. It is based on logical clock, and it combines transactions and pages. GSN makes a synchronization only when a set of operations have dependencies, and thus it provides high performance for independent operations.

Works in the second category include Silo [7][9] and FOEDUS [3]. They do not use the global sequence number. They combines the epoch and the local sequence number. During an epoch (typically 10 to 40 ms), conflicted transactions are aborted. At the end of an epoch, non conflicted transactions are provided the log sequence numbers, and then they are transferred to storage devices in a lump. The concurrency protocol is optimistic concurrency control, and it performs efficiently in this scheme. One disadvantage of this scheme is a relatively long latency since all the transactions in an epoch are urged to wait until the end of the epoch. This scheme is expected to provide dramatically high performance. However, since this design scheme is totally different from that of a conventional DBMS, its adoption for a conventional DBMS would require huge re-implementation effort, while our proposed scheme is easy to be implemented.

## VIII. Conclusions

This paper addresses to accelerate WAL by exploiting ioDrive. The ioDrive shows an interesting feature. Non-contiguous parallel write operations show higher performance compared with sequential writes, which never happens in a hard disk drive. We implement a prototype in-memory transaction processing system, and evaluate the performance of P-WAL with it. P-WAL achieves 425,531 tps which is around 2.42 times higher compared with the conventional method. On the other hand, the improvement on TPC-C is quite limited.

In future work, we first investigate whether P-WAL fully uses the IO bandwidth of ioDrive. If we find that there is a chance of optimization through the analysis, then we will try the optimization. Second, we try to implement P-WAL onto a real DBMS such as PostgreSQL.

## Acknowledgement

This work is partially supported by JST CREST "System Software for Post Petascale Data Intenseive Science" and JST CREST "Extreme Big Data (EBD) Next Generation Big Data Infrastructure Technologies Towards Yottabyte/Year". It is also supported by JSPS KAKENHI Grant Numer 25280043HA and JST CREST "Statistical Computational Comsmology with Big Astronomical Imaging Data".

## References

[1] Harizopoulos, S., Abadi, D. J., Madden, S. and Stonebraker, M.: OLTP through the looking glass, and what we found there, SIGMOD, pp. 981–992 (2008).

[2] Johnson, R., Pandis, I., Stoica, R., Athanassoulis, M. and Ailamaki, A.: Aether: A Scalable Approach to Logging, PVLDB, Vol. 3, No. 1, pp. 681–692 (2010).

[3] Kimura, H.: FOEDUS: OLTP Engine for a Thousand Cores and NVRAM, SIGMOD Conference (2015).

[4] Levandoski, J. J., Lomet, D., Mokbel, M. F. and Zhao, K.: Deuteronomy: Transaction Support for Cloud Data, CIDR, pp. 123–133 (2011).

[5] Mohan, C., Haderle, D. J., Lindsay, B. G., Pirahesh, H. and Schwarz, P. M.: ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging, ACM Trans. Database Syst, Vol. 17, No. 1, pp. 94–162 (1992).

[6] NYSE: NYSE, New York Stock Exchange > About Us > News & Events > News Releases > Press Release 06-03-2009:, http://www1.nyse.com/press/1244024115279.html. （Accessed: 2015-04-15）.

[7] Tu, S., Zheng, W., Kohler, E., Liskov, B. and Madden, S.: Speedy transactions in multicore in-memory databases, SOSP, pp. 18–32 (2013).

[8] Wang, T. and Johnson, R.: Scalable Logging through Emerging Non-Volatile Memory, PVLDB, Vol. 7, No. 10, pp. 865–876 (2014).

[9] Zheng, W., Tu, S., Kohler, E. and Liskov, B.: Fast Databases with Fast Durability and Recovery Through Multicore Parallelism, OSDI, pp. 465–477 (2014).