International Journal of Networking and Computing – www.ijnc.org, ISSN 2185-2847
Volume 9, Number 2, pages 339–353, July 2019

Integration of TicToc Concurrency Control Protocol with Parallel Write Ahead Logging Protocol

Yasuhiro Nakamura
Graduate School of Systems and Information Engineering, University of Tsukuba
Tsukuba, Ibaraki, 305-8577, Japan

Hideyuki Kawashima
Faculty of Environment and Information Studies, Keio University
Fujisawa, Kanagawa, 252-0882, Japan

Osamu Tatebe
Center for Computational Sciences, University of Tsukuba
Graduate School of Systems and Information Engineering, University of Tsukuba
Tsukuba, Ibaraki, 305-8577, Japan

Received: February 15, 2019
Revised: May 6, 2019
Accepted: June 11, 2019
Communicated by Takeshi Ohkawa

Abstract

A transactional system consists of a concurrency control system and a recovery system. TicToc is one of the state of the art concurrency control protocols today, but it lacks recovery system. We studied the ways to integrate TicToc and recovery system. For efficiency, we adopted a parallel write ahead logging scheme for the recovery system. There are two methods to optimize the logging. First method is early lock release which executes lock release early on data objects. Second method is group commit which executes batched logs transfer to storage from memory. We integrated a transactional system consisted by TicToc and P-WAL logging system assuming non-volatile memory. We found that the two optimization methods incur performance degradation when storage access latency is equivalent to that of NVRAM.

Keywords: Concurrency Control, Write Ahead Logging, Transaction Processing, Database System

1 Introduction

1.1 Background

To manage data in an information system today, transaction processing is mandatory. Transaction provides four properties: atomicity, consistency, isolation and durability, which is abbreviated as ACID. To preserve ACID properties, transaction processing requires two key modules: **concurrency control** and **recovery** [19]. Without concurrency control, concurrent execution of transactions violates the isolation property. Without recovery, transactional systems do not successfully restart after system failures. Therefore, both modules are essentially important for any transaction processing systems.

The acceleration of transaction processing has been intensively studied. Recent advance on hardware provides researchers chances to re-design the system. Using many-core CPU and parallel I/O storage (e.g.

flash device, non-volatile memory (NVRAM)), new transactional systems have been developed today. Some work consider the whole system [9, 15], while some work consider only either concurrency [4, 12, 21] or recovery [7,8,17].

1.2 Question

TicToc [21] is one of the most efficient concurrency control modules today. It is based on optimistic concurrency control protocol, and demonstrates better performance than Silo [15]. However, the original paper only presents concurrency control protocol, and it does not describe how to appropriately integrate recovery module to TicToc. To make TicToc a transactional system, the recovery module should be integrated in a sophisticated fashion.

The performance bottleneck of recovery module is the write ahead logging (WAL). A paper reports that WAL requires 11.9% of CPU cycles in a typical workload called TPC-C [5]. To accelerate WAL, parallelized WAL protocols are recently proposed [8, 17, 22].

How should we integrate TicToc with such parallel logging protocols? These recent protocols adopt two optimization methods: **group commit** and **early lock release**. Group commit [6] synchronizes multiple log records from memory to storage device simultaneously to improve I/O bandwidth utilization. Early lock release [7] conducts lock release before synchronizing logs from memory to storage. It shortens the time for the block of tuples incurred by lock acquisition. Modern transactional systems adopt the both of methods [8, 17, 22], and any counterexamples are not reported yet.

1.3 Contribution

In this paper, we report a counterexample. When integrating TicToc with parallel write ahead logging P-WAL [8], the adoption of optimization methods degrades performance. This is due to the fact that TicToc requires a shared counter that is concurrently accessed by multiple worker threads frequently. Besides P-WAL, there are other parallel logging protocols [17, 22]. Their key feature is parallelized synchronization of log records, and it is in common with P-WAL.

We also found that the degradation phenomenon does not occur with a conventional concurrency protocol such as strict two phase locking [19]. This phenomenon also does not occur for another state of the art concurrency protocol Cicada [12] as reported in [14]. To our knowledge, this paper reports the first counterexample.

1.4 Organization

The rest of this paper is organized as follows. Section 2 describes the concurrency control protocol of TicToc. Section 3 describes a logging protocol called P-WAL [8], and describes optimization methods of parallel logging protocol. Section 4 describes four ways to integrate P-WAL with TicToc. Section 5 describes design and implementation of the integrated system. Section 6 evaluates four patterns of implementation because each method has two parameters and we have two optimization methods. Section 7 describes related work. Finally, Section 8 describes conclusions.

2 TicToc: Concurrency Control Protocol

Recent studies [12, 15, 21] have unveiled the potential of optimistic concurrency control (OCC) originally proposed in decades ago [11]. The protocol is divided into three phases: **read phase**, **validation phase**, and **write phase**. The flow of this process is shown in Algorithm 1. First, the read phase is executed. The worker copies the tuple to be operated from the database to the local area and performs the operation on it. At this time, the tuple that read was set to the read set, and the tuple where the write was performed is held in the write set. Then it verifies in the validation phase whether the result of operation keeps consistency with other workers. If consistency is maintained, the worker reflects the tuple change in the database in the write phase. When inconsistency is detected, the transaction is aborted, and restarts from the read phase.

Conventional OCCs use a shared counter to obtain the timestamp of a transaction, which is a performance bottleneck which is intensively analyzed in [20]. In TicToc [21], in order to eliminate the bottleneck, the timestamp is calculated in a decentralized way. Two types of timestamps are managed in TicToc. A tuple has **write timestamp (wts)** and **read timestamp (rts)**. These are the time stamp in which the value of the tuple was written and the time stamp last read, respectively. The tuple has wts and rts because the value of the tuple indicates the valid range. In order to atomically read wts and rts, TicToc keeps them in one 64 bit word. This is called **timestamp word (TS word)**.

Algorithm 1 Optimistic Concurrency Control

```
Retry point:
readPhase()
doOperation()
if validationPhase() is fail then
    abort()
    retry()
end if
writePhase()
```

Read phase copies data from database to local area. A transaction reads the TS word twice to atomically obtain the timestamp and value of the tuple. If there was no change in TS word, we copied the tuple to the local area of the worker thread because we were able to get the correct combination of timestamp and value. If the tuple is locked, the operation on the tuple is incomplete and there is a possibility that the correct combination of timestamp and value can not be obtained. Therefore, reading is repeated until the lock is released.

Validation phase validates consistency. To calculate the timestamp, use the tuple (local tuple) copied to the local area of the worker in the read phase. The timestamp is the maximum value of wts of the local tuple in the read-set and rts + 1 of the tuple in the write set. Based on the calculated timestamp, the validity of the transaction is determined as follows. If the timestamp is less than or equal to rts of the local tuple in the read set, inconsistency will not occur since the tuple is guaranteed not to change from wts to rts. If the timestamp exceeds rts, it checks whether wts of the tuple in the database is equal to wts of the local tuple. If they are equal, the tuple has not changed since it was copied in the read phase, so no inconsistency has occurred. At this time, in order to guarantee that the tuple does not change until the commit timestamp, rts of the tuple are extended to the timestamp. If wts are different, abort this transaction because another worker thread is changing the tuple. Since tuples in the write set acquire the lock at the beginning of the validation phase, no confirmation is necessary.

Write phase copies data from local area to database. In the write phase, only transactions that passed the validation phase and are determined to be inconsistent execution in the write phase. The transaction updates the value and timestamp of the data object. The transaction then releases the lock on the tuple.

3 Parallel Logging Protocol

3.1 P-WAL

To conduct recovery, database modifications needs to be logged in stable storage during normal processing. This operation is called logging, and its acceleration is important to improve system performance [5].

P-WAL [8] was proposed as a logging protocol suitable for NVRAM or flash device that involve inherent parallelism. The architecture of P-WAL is shown in the right-hand side of Fig. 1. Different from serial-WAL used in practical databases (e.g. PostgreSQL, MySQL), P-WAL isolates both of the WAL buffer and the WAL file by providing them for each worker thread. By the isolation, worker threads are unleashed from contentions with lock acquisitions.

Conventionally, the WAL buffer realized sequential access to the HDD, so there is only one buffer in the system as shown in left-hand side of Fig. 1. As a result, contention occurred when inserting log records into the WAL buffer, causing performance degradation. In a device where parallel random access has higher performance than a single sequential access, it is faster to write logs in parallel. Therefore, making each

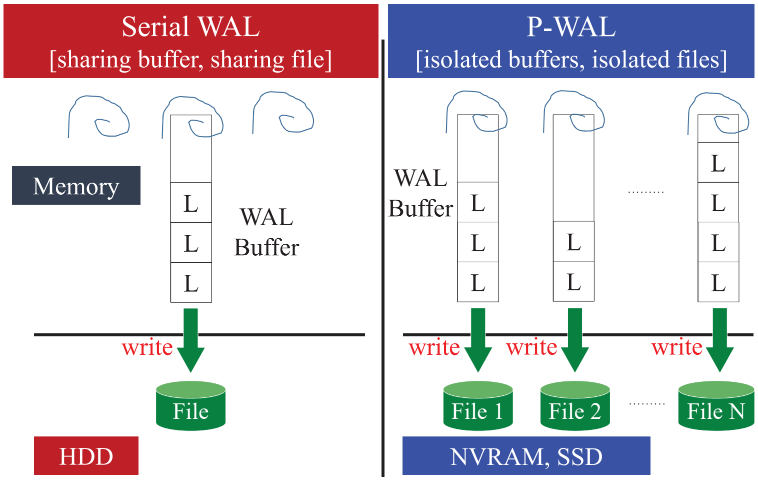


Figure 1: P-WAL Overview

worker thread have a WAL buffer and writing to each corresponding WAL file is a structure of WAL suitable for flash storage.

Because P-WAL provides a WAL buffer for each worker thread, the order of the logs becomes ambiguous. Therefore, in P-WAL, the log sequence number (LSN) is issued to each log record using a single shared counter. LSN increases incrementally with each issuance. Since this shared counter needs to be manipulated atomically, it needs to be realized by an atomic operation such as fetch-and-add or compare-and-swap.

Even if log records are persisted, that transaction is not always committable. This is because if the transaction depends on log records generated by other worker threads, that log record is not persisted, it does not satisfy the commit condition of the transaction. In order to control the notification of the commit, each WAL buffer holds the LSN (flushed LSN) of the log record that it last persisted. The commit condition of a certain transaction is the case where the minimum one of the flushed LSNs held by all the WAL buffers is equal to or larger than the own LSN. The worker thread keeps queued transactions waiting to be committed and checks the flushedLSN of the other WAL buffer each time the transaction ends. Then, it notifies the commit of the transaction that satisfies the commit condition, and excludes the transaction from the queue.

Besides P-WAL, there are other parallel logging protocols [17, 22]. Their key features is parallelized synchronization of log records and notification control, which are in common with P-WAL.

3.2 Optimization Methods

3.2.1 Early Lock Release

In the concurrency control protocol, locks are acquired before data access. There are **conservative lock release (CLR)** and **early lock release (ELR)** [7] when releasing this lock. CLR is a method of releasing the lock of the data object after persisting the log record. TicToc does not read locked tuples in read phase. Therefore, you can guarantee that the value of the data object that the worker thread reads in the read phase is persisted.

ELR is a method of releasing locks on data objects before persisting log records. This makes it possible to shorten the lock holding period by the latency required for log record persistence. By releasing the lock early, ELR relaxes conflicts with other worker threads and improves concurrency. ELR releases the lock on the data object before persisting the log. At this point there is no guarantee that all data on which the transaction depends is being persisted. Therefore, it gives the total order to the log record and checks whether the log record before the log generated by the transaction to be notified is persisted. When all dependent logs are made persistent and it is confirmed that the system can be recovered, notification control notifies the client of the commit.

To issue this total sequence number, there is a method of using a single log buffer and a method of issuing a log sequence number (LSN) using a single shared counter.

3.2.2 Group commit

Group commit [6] is a method of persisting multiple log records collectively across multiple transactions. By collectively persisting log records, I/O bandwidth usage to the persistent device is made more efficient and transaction processing performance is improved.

Group commit is applicable to both the CLR and the ELR. When group commit is applied to the CLR, it is necessary to hold the lock across multiple transactions because it is necessary to release the lock after persisting the log record. It is known that deadlock does not occur when acquiring locks according to the total order of data objects. However, when applying group commit to CLR, deadlock can occur between worker threads.

4 Integrating TicToc with P-WAL

For the system to be evaluated, TicToc was used for the concurrency control method and P-WAL was used for the log writing method. We designed the following four protocols to compare the effect on performance by using ELR and group commit.

1. `CLR:NoGroup` applies conventional lock release without group commit.

2. `CLR:Group` applies conventional lock release with group commit.

3. `ELR:NoGroup` applies early lock release without group commit.

4. `ELR:Group` applies early lock release with group commit.

4.1 Design of `ELR:Group` and `ELR:NoGroup`

CLR and ELR show the difference in lock release timing, the former is CLR and the latter is the protocol adopting ELR. NoGroup and Group indicate the presence or absence of group commit, the former does not commit group, the latter is the group commit protocol.

To efficiently integrate P-WAL with TicToc, we need to consider the assumption with both methods in P-WAL. Since P-WAL assumes ELR and group commit, `ELR:Group` is naturally designed. By reducing the number of waiting transaction in group commit, we can design `ELR:NoGroup`.

It should be noted that P-WAL requires a single shared counter to generate the log sequence number. This is implemented with an atomic operation, the fetch-and-add, and described as "$tx.lsn \leftarrow$fetchLSN()" in Algorithm 2 and 3. This counter has some negative effects on performance as will be seen in Section 6.

The procedure of `ELR:Group` and `ELR:NoGroup` are are shown in Algorithm 2 and 3 respectively.

4.2 Design of `CLR:NoGroup`

Integration with CLR is not straightforward. Since P-WAL assumes ELR, LSN is assigned to each log using a shared counter. This is to solve the notification control and the order of log application at the time of recovery. However, using the shared counter contravenes TicToc's design philosophy, which thoroughly eliminated the shared area.

Therefore, utilization of LSN may sacrifice high concurrency of TicToc. TicToc does not read locked tuples in read phase. Therefore, when CLR is used, it can be guaranteed that the value of the data object that the worker thread reads in the read phase is persisted.

Because of this nature, notification control is unnecessary in CLR, and if the timestamp generated by TicToc is added to the log record, the order of log application at the time of recovery can be resolved. ELR can not use timestamp of TicToc like CLR. TicToc's timestamp is calculated using only tuple's wts and rts.

Since the timestamp does not monotonically increase and since there are a plurality of identical timestamps, it is impossible to judge in the notification control whether or not the log record before the log generated by the transaction to be notified is persisted.

In this protocol, when ELR is used, LSN is issued for each transaction, and it is used for notification control and recovery. When CLR is used, the timestamp generated by TicToc is used for recovery. The procedure of `CLR:NoGroup` is shown in Algorithm 4.

4.3 Design of `CLR:Group`

When group commit is applied to CLR, deadlock may occur between worker threads. In this protocol, a method of eliminating the circulation standby without using a timer was used. The deadlock occurs by waiting for circulation to the lock. There are two processes to wait for locking with this protocol.

The first one is waiting to acquire the lock in the validation phase. When failing to acquire the lock, if you release all the locks of the worker like No-Wait Locking, the deadlock does not occur. In order not to deviate from the CLR method, we make the WAL buffer persist before lock release.

The second is when waiting to release locks when copying tuples in the read phase. In two worker threads, deadlock occurs if the value you want to copy from the database is locked by the other party. Therefore, when lock release wait occurs at the read phase, the WAL buffer is made to be perpetual like the first process, and all locks of the worker are released. The procedure of `CLR:Group` is shown in Algorithm 5.

Algorithm 2 `ELR:Group`

```
Require: WAL Buffer $walBuffer$, Group Commit Number $N$
$ntx \leftarrow 0$;
$flushedLSN \leftarrow 0$;
$commitQueue \leftarrow <>$;                                   # <> is empty queue
while runable() do
    $tx \leftarrow$fetchTransaction();
Retry point:
    readPhase(); doOperation();
    if validationPhase() is fail then
        abort(); retry();
    end if
    $tx.lsn \leftarrow$fetchLSN(); writePhase(); releaseLocks(); $commitQueue$.push(tx); $ntx \leftarrow ntx + 1$;
    if $ntx == N$ then
        $flushedLSN \leftarrow walBuffer$.flush(); $ntx \leftarrow 0$;
    end if
    $minFlushedLSN \leftarrow$ min($worker.flushedLSN \mid worker \in workers$);
    while $commitQueue$.notEmpty() do
        $t \leftarrow commitQueue.front$;
        if $minFlushedLSN \geq t.lsn$ then
            reply($t$); $commitQueue$.pop();
        else
            break;
        end if
    end while
end while
```

5 System Implementation

We design and implement a system for evaluation. The prototype system adopted the stored procedure method. That is, each worker has an instruction sequence procedure to be executed. Procedure is executed by interacting with Database via Transaction manager. The Transaction manager controls the entire transaction

Algorithm 3 `ELR:NoGroup`

```
Require: WAL Buffer $walBuffer$
$flushedLSN \leftarrow 0$; $commitQueue \leftarrow <>$;                    # <> is empty queue
while runable() do
    $tx \leftarrow$fetchTransaction();
Retry point:
    readPhase(); doOperation();
    if validationPhase() is fail then
        abort(); retry();
    end if
    $tx.lsn \leftarrow$fetchLSN(); writePhase(); releaseLocks(); $commitQueue$.push(tx); $flushedLSN \leftarrow$ $walBuffer$.flush();
    $minFlushedLSN \leftarrow$ min($worker.flushedLSN \mid worker \in workers$);
    while $commitQueue$.notEmpty() do
        $t \leftarrow commitQueue.front$;
        if $minFlushedLSN \geq t.lsn$ then
            reply($t$); $commitQueue$.pop();
        else
            break;
        end if
    end while
end while
```

Algorithm 4 `CLR:NoGroup`

```
Require: WAL Buffer $walBuffer$
while runable() do
    $tx \leftarrow$fetchTransaction();
Retry point:
    readPhase(); doOperation();
    if validationPhase() is fail then
        abort(); retry();
    end if
    $walBuffer$.flush(); writePhase(); releaseLocks(); reply($tx$);
end while
```

Algorithm 5 `CLR:Group`

```
Require: WAL Buffer $walBuffer$, Group Commput Number $N$
$ntx \leftarrow 0$;
$txQueue \leftarrow <>$;                                                   # <> is empty queue
while runable() do
    $tx \leftarrow$fetchTransaction()
Retry point:
    readPhase(); doOperation();
    if validationPhase() is fail then
        abort(); $walBuffer$.flush(); releaseLocks(); reply($txQueue$); $txQueue \leftarrow <>$; retry();
    end if
    writePhase(); $txQueue$.push($tx$); $ntx \leftarrow ntx + 1$;
    if $ntx == N$ then
        $walBuffer$.flush(); releaseLocks(); reply($txQueue$); $txQueue \leftarrow <>$;
    end if
end while
```

process. WalBuffer receives and accumulates log records from the Transaction manager, and writes the accumulated log records to the corresponding WAL File by the control command from the Transaction manager to persist the log records.

We used the programming language C++ to implement the system. Procedure registered by hard coding to the system. The column definition of the table is described by a C++ structure, and each table is represented by an array of structures. The array was fixed length, and no new element was inserted. The arguments given to the procedure were generated using the pseudo random number xorshift 128+ [2].

When writing to WAL File, write to NVRAM or SSD is assumed, not write to actual storage. Assumed latencies for NVRAM or SSD are constant. Note that during transaction execution, only writing to NVRAM or SSD occurs, and reading from NVRAM occurs only during recovery, so reading latency does not need to be considered in this experiment. For this latency insertion processing, the Time Stamp Counter (TSC) of the CPU was used. Monitor the TSC and wait for the CPU clock to advance by the specified standby time.

6 Evaluation

We evaluate the four protocols proposed in Section 4. We first describe result for a case with low parallelism using Intel E5 in Section 6.1, and then describe result for a case with low parallelism using Intel Xeon Xeon Phi in Section 6.2.

There are some limitations in this evaluation. First, the prototype database system does not provide index structures such as Masstree [13]. Since index traversal requires some more cost rather than direct array access, the result of performance is considered to be ideal in the viewpoint of access method. Second, we emulate storage latency by putting delay using RDTSC operation, and we did not use any storage emulator because we do not find any emulators which are appropriate for NVRAM. Quartz [16] is an excellent simulator, but it does not support write latency emulation. Although, the RDTSC emulation approach is adopted in some studies in database field [9], the usage of appropriate emulator would improve the quality of study, and it is left as future work.

6.1 Small Scale Case with Xeon E5

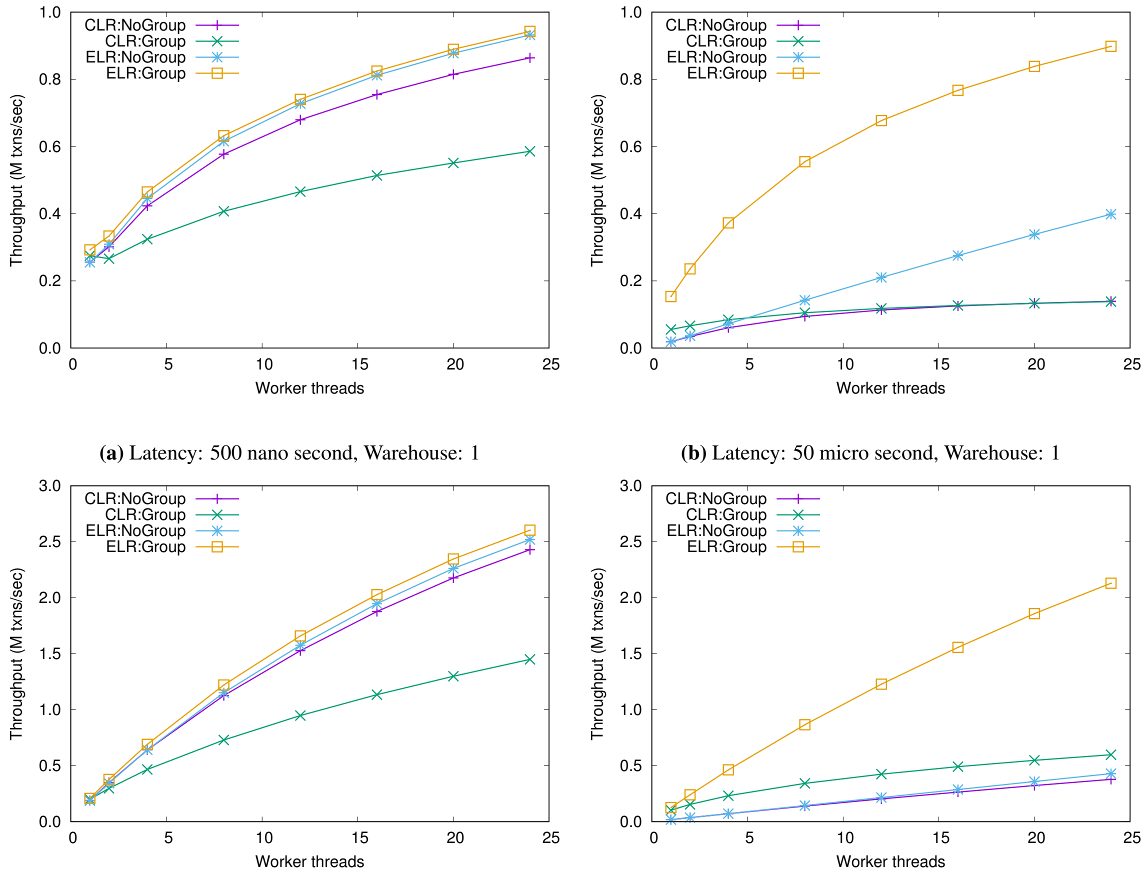
In the experiment, throughput was measured by changing the storage latency of warehouse and WAL. The number of warehouses was changed to 1 and 10. Storage latency was changed to 500 nano second and 50 micro second. The experimental environment is shown in Table 1. The size of group commit was set to 16, which means log records for 16 transactions are synchronized simultaneously.

Table 1: Environment

Processor	Intel(R) Xeon(R) CPU E5-2695 v2 @ 2.40GHz
Physical Cores	12 cores
Logical Cores	24 cores
RAM	64GB
OS	CentOS release 6.8 (Final)

The experiment for 1 warehouse is shown in Figure 2a and 2b. `ELR:Group` shows the best performance in all the cases. Comparing the two methods of ELR, it can be seen that the performance difference increases as the storage latency increases. When the storage latency is 500 nano second, there is almost no performance difference between `ELR:Group` and `ELR:NoGroup`, but when storage latency is 50 micro second, `ELR:Group` has 2.25 times better throughput than `ELR:NoGroup`. Therefore, as the write latency to the WAL file is larger, higher performance is achieved by group committing, and with a small latency of about 500 nano second, almost no performance improvement can be obtained with group commit.

The experimental results at 10 warehouses are shown in Fig. 2c and 2d. Comparing this result with 1 warehouse, the trend of throughput change due to the change in the number of workers has not changed much. However, as the number of warehouses increases, the degree of dispersion increases and the throughput improves overall. Focusing on the case where the storage latency is 500 nano second, the throughput of



(a) Latency: 500 nano second, Warehouse: 1

(b) Latency: 50 micro second, Warehouse: 1

(c) Latency = 500 nano second, Warehouse: 10

(d) Latency = 50 micro second, Warehouse: 10

Figure 2: TPC-C Result (Intel E5)

CLR:NoGroup approaches the throughput of ELR as the number of warehouses increases. This is thought to be due to the fact that the conflict of the lock on the data object has decreased due to the increase in the degree of dispersion.

6.2 Large Scale Case with Xeon Phi

We evaluate the performance of the four methods CLR:NoGroup, CLR:Group, ELR NoGroup, ELR:Group which combine CLR/ELR and group commit respectively by experiment with Intel Xeon Phi that has 68 cores. The experimental environment is shown in Table 2. The size of group commit was set to 16.

In the experiment, the warehouse parameter of TPC-C benchmark was set to 64, the storage latency was changed with RDTSC. The storage latency parameter was changed to 500 nano second or 50 micro second. The 500 nano second latency assumes DIMM type NVRAM device, and the 50 micro second latency assumes SSD device.

6.2.1 Result

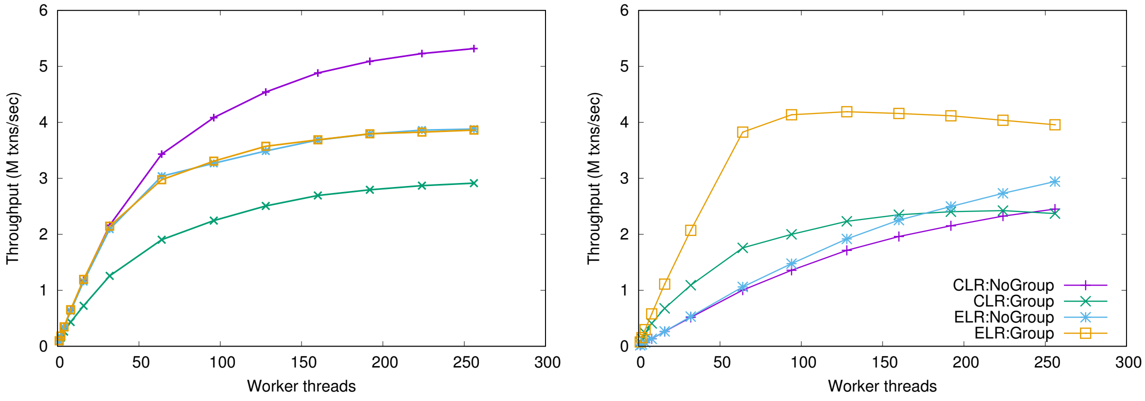
Impact of Storage Latency The result of TPC-C benchmark is shown in Fig. 3a and Fig. 3b. For storage latency of 50 micro second (SSD), the ELR:Group shows the best performance. It is considered that the optimization methods are performed effectively.

However, with storage latency of 500 nano second (NVRAM), CLR:NoGroup, which does not use any optimization methods, shows the best performance. In addition, ELR:Group and ELR:NoGroup have the

Table 2: Environment

CPU	Intel(R) Xeon Phi(TM) CPU 7250 @ 1.40GHz
#Physical Core	68
#Logical Core	274 cores
RAM	MCDRAM 16GB DRAM 96GB
OS	CentOS Linux release 7.2.1511 (Core)
Storage	Latency Emulation with RDTSC
Benchmark	TPC-C

similar performance. It is considered that high performance by group commit and ELR are not obtained in this case.



(a) Storage Latency: 500 nano second

(b) Storage Latency: 50 micro second

Figure 3: Non Optimized Method `CLR:NoGroup` Wins (warehouse: 64, latency: 500 ns)

Impact of Group Commit on CLR We evaluated how the group commit size affects performance when the latency of WAL writing is small. The group commit size was changed and measured with CLR and ELR, respectively. The number of warehouses was set to 64 and storage latency was set to 500 nano second. The experimental results are shown in Fig. 4.

From the figure, it is observed in CLR that an increase in group commit size results in performance degradation. In the ELR, even if the group size was changed, no performance change was seen. In CLR, lock acquisition time increases as group commit size increases, so it is considered that the performance has deteriorated due to an increase in aborts accompanying collision. In ELR, the lock acquisition time is short regardless of the size of the group commit size since it releases locks before synchronizing logs. So, it is considered that the performance did not deteriorate.

Comparison with S2PL It should be noted that this phenomenon is expressed in an excellent concurrency control method. Figure 5a shows the results when the concurrency control method is set to Strict Two-Phase Locking (S2PL) [19]. As can be seen from the figure, since the performance is low in S2PL, the phenomenon observed with TicToc is not developed.

6.2.2 Effect of LSN Access

From the experimental result shown in Fig. 3a, it is considered that the issue of LSN is caused by the performance of `ELR:Group` and `ELR:NoGroup` being lower than that of `CLR:NoGroup`. In the many core environment, issuance of LSN becomes expensive, and it seems that probably the lock holding period

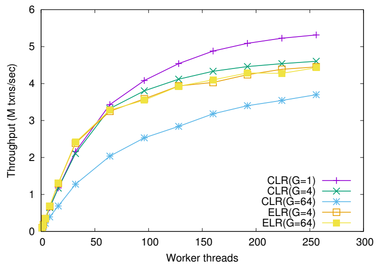
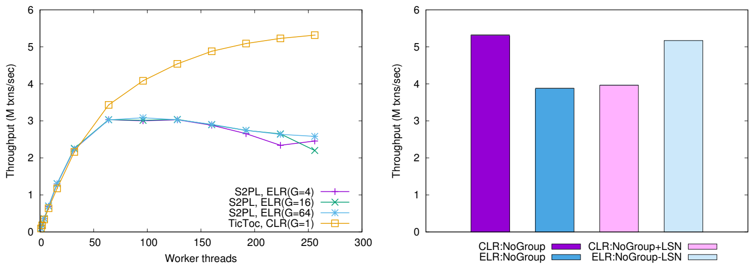


Figure 4: Effect of Group Commit



(a) Comparing TicToc with S2PL

(b) Effect of LSN Access

Figure 5: Micro Analyses

becomes longer than the CLR which inserts storage latency every 500 nano second every time. Therefore, we implemented `CLR:NoGroup+LSN` which issued LSN also in `CLR:NoGroup` and `ELR:NoGroup-LSN` which did not issue LSN in `ELR:NoGroup` and measured its performance. Note that `ELR:NoGroup-LSN` is not recoverable.

Fig. 5b shows the comparison of the throughput of each method in 256 threads. `CLR:NoGroup+LSN` to the same extent as `ELR:NoGroup` deteriorated. On the other hand, `ELR:NoGroup-LSN` improved performance to the same extent as `CLR:NoGroup`. Therefore, we think that the performance of ELR became lower than CLR is due to the issuance of LSN.

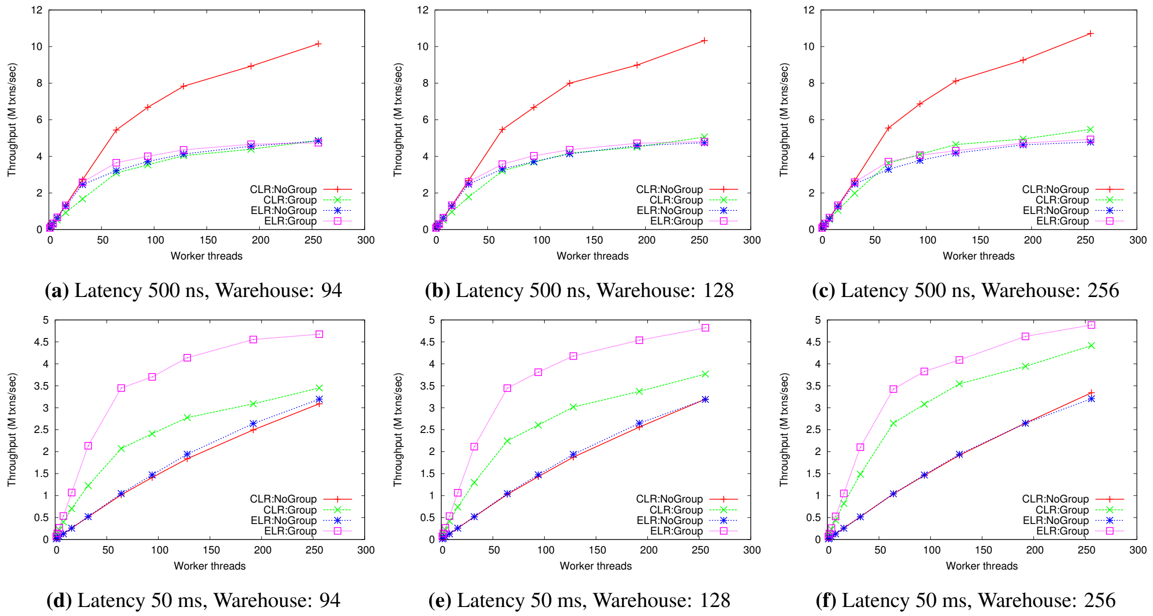
6.2.3 Effect of Warehouse Size

To understand the behavior of four parameter settings, we conducted TPC-C benchmark varying the number of warehouses that is related to the probability of conflicts for database objects. The smaller the warehouse, the higher the conflict probability. The larger the warehouse, the smaller the conflict probability.

Large Number of Warehouses Figure 6 show the result of TPC-C when warehouse sizes are relatevely large (94, 128 and 256). The results with NVRAM latency (Figure 6a, 6b and 6c) are similar to that of the 64 warehouse case shown in Figure 3a. As the size of warehouses increase, the probability of conflicts reduces in TPC-C benchmark. Therefore, the larger the size of the warehouse, the better the performance. We observe the same results on SSD latency (Figure 6d, 6e and 6f). The reason why three figures (Figure 6a, 6b and 6c) show almost the same result is that all of parameter settings achieve the best performances on when scale is 94, and more conflict relaxation does not affect the performances.

Small Number of Warehouses Figure 7 show the result of TPC-C when warehouse sizes are relatevely small (1, 2 and 4). Figure 7a, 7b and 7c show that performance of `CLR:NoGroup`, `ELR:NoGroup` and `ELR:Group` are compatible, and all of them are better than group commit based settings (`CLR:Group`. Because the size of warehouses are relatively small, many conflicts occur with database accesses. In this case, the adoption of group commit in `CLR:Group` increases the number of aborts. This is because our protocol adopts the no-wait policy that immediately aborts for a transaction if it finds that its target database object is already locked by another transaction. Figure 7d, 7e and 7f show that `ELR:NoGroup` is more scalable than `ELR:Group`. This is because `ELR:NoGroup` does not use the centralized counter to issue the log sequence number (i.e. fetch_and_add instruction). Such centralized counter degrades performance in highly concurrent situation.

It also should be noted that the peak performance of `ELR:Group` is better than that of `ELR:NoGroup` in Figure 7d, 7e, 7f. This is due to the fact that storage access latency for these experiments are set to be large (50 micro second) by supposing SSD device, and `ELR:NoGroup` issues more synchronization operations than that of `ELR:Group`. This increased execution time of each transaction, and therefore throughput was degraded.



(a) Latency 500 ns, Warehouse: 94

(b) Latency 500 ns, Warehouse: 128

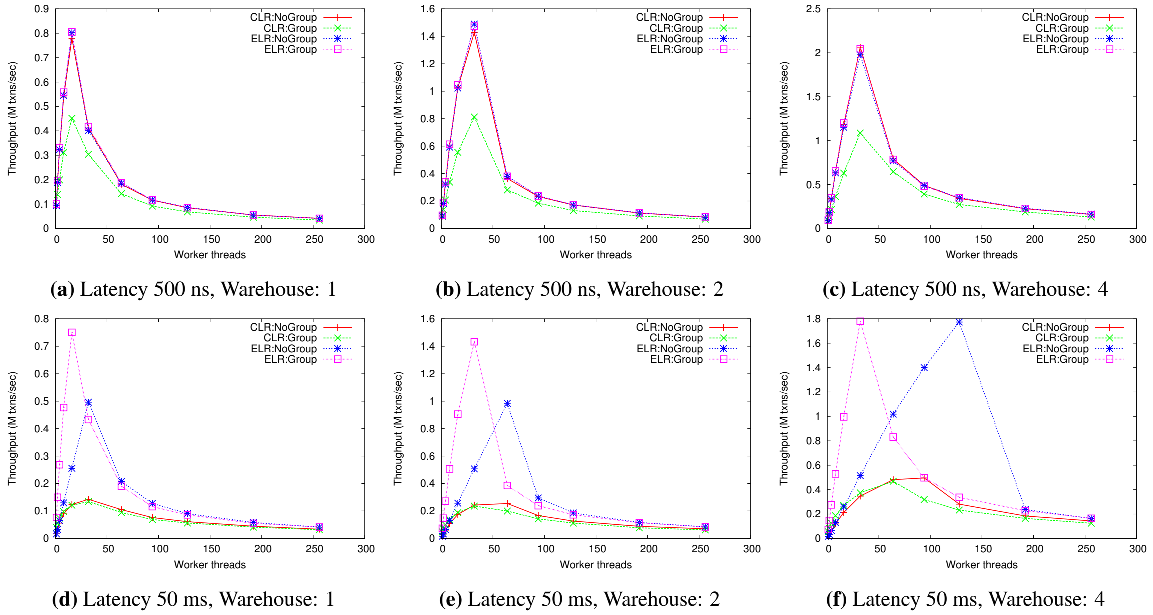
(c) Latency 500 ns, Warehouse: 256

(d) Latency 50 ms, Warehouse: 94

(e) Latency 50 ms, Warehouse: 128

(f) Latency 50 ms, Warehouse: 256

Figure 6: TPC-C Result (Warehouse: 94, 128, 256)



(a) Latency 500 ns, Warehouse: 1

(b) Latency 500 ns, Warehouse: 2

(c) Latency 500 ns, Warehouse: 4

(d) Latency 50 ms, Warehouse: 1

(e) Latency 50 ms, Warehouse: 2

(f) Latency 50 ms, Warehouse: 4

Figure 7: TPC-C Result (Warehouse: 1, 2, 4)

7 Related Work

7.1 Modern Transactional System

From the viewpoint of transactional system, we describe some modern concurrency control protocols and logging protocol here. After Silo [15, 22], a bunch of studies are published.

7.1.1 Concurrency Control and Recovery

Silo [15, 22] is an in-memory database system created to achieve excellent performance and scalability on multi-core machines. Among them, a time stamp calculation method and a commit protocol are proposed. By notifying the completion of commit for each unit called Epoch, throughput is increased instead of sacrificing latency. A shared counter is used to identify Epoch. This counter is periodically incremented by a dedicated thread, and synchronization processing is performed by each worker thread reading this value.

FOEDUS [9] is a transaction processing system designed on the premise of nonvolatile memory. In order to improve the performance of range search, MasterTree that Masstree [13] corresponded to nonvolatile memory is newly proposed, and the commit protocol and concurrency control method based on Silo [15] are proposed. As an application of FOEDUS, a graph processing system Janus [10] has been devised. In addition, high performance concurrency control method MOCC [18] combining optimistic execution control and pessimistic execution control using temperature control is implemented on FOEDUS.

7.1.2 Concurrency Control

TicToc [21] discussed in this research is a time stamp based concurrency control method, which is reported to exhibit higher performance than Silo. On the other hand, as mentioned in this research, the performance in the many core environment is unknown. Furthermore, the integration method with the parallel logging method has not been studied so far, to our knowledge.

Cicada [12] is a concurrency control mechanism that combines multi-version concurrency control (MVCC), optimistic concurrency control (OCC), and distributed timestamp generation scheme. MVCC makes it possible to avoid competition between read and write locks by creating a new version whenever data is updated. OCC provides a method of transaction control that emphasizes efficiency on the expectation that transactions will not compete, and it consists of three phases: read, validation, and write. The protocol executes an operation in the read phase without holding any locks, maintains consistency in the validation phase, and the reflects the update in the database in the write phase. Distributed timestamp generation is a technique for generating timestamp allocated to versions for sequence mediation in multiple worker threads assigned in different CPU cores. Cicada exhibits novel performance when data accesses by transactions are extremely skewed.

7.1.3 Recovery

Wang et al. [17] proposes a passive group commit, which is a method of preparing multiple WAL buffers in NVRAM and transferring log records in parallel to them. Passive group commit places WAL buffer on NVRAM and writes log directly to NVRAM without going through DRAM. Passive group commit is ELR, and a dedicated daemon is prepared to check whether each transaction satisfies the commit condition collectively. Also, a passive group commit requires a sequence number called Global Sequence Number.

P-WAL [8] is a parallel log precedence writing method that occupies log buffer and log file for each worker thread. In the original paper, it is stated that P-WAL shows better performance than Aether [7] on ioDrive. The difference between P-WAL and passive group commit is handling of DRAM. P-WAL is also applicable to SSD, io-Drive etc. To install WAL buffer on DRAM, while passive group commit writes log record directly to NVRAM.

7.2 Integration of Concurrency Control with Logging

There are some work on the integration of modern concurrency control protocols and modern logging protocols because some papers only describes either of them, but both of them are required to make a transactional system complete. In this paper we tried to make TicToc [21] complete, while another paper authors try to make Cicada concurrency control protocol [14] complete by integrating a logging protocol.

In the paper [14], authors show that the use of two optimization methods (WAL and ELR) provides performance improvement. This is due to the fact that the log sequence number (LSN) can be implemented by using distributed timestamp counter that is provided in Cicada. The counter is originally used to decide the order of multiple transactions executed in different CPU cores. Typical method requires a single shared counter for this purpose, but concurrent accesses to a single object degrades performance dramatically even if atomic operation (e.g. fetch-and-add) is adopted. Cicada avoids this problem by introducing a distributed timestamp generation scheme. By leveraging this scheme to log sequence number generation for the logging protocol, the LSN access cost problem is avoided.

In case of TicToc, the LSN access cost problem exists when applying the ELR optimization as shown in Figure 5b. This is because TicToc requires a single shared counter when applying ELR which is described as "$tx.lsn \leftarrow$fetchLSN()" in Algorithm 2 and 3.

8 Conclusions

In this paper, we proposed a method to efficiently integrate the concurrency control method TicToc and parallel write ahead logging method P-WAL. In order to streamline the write ahead logging method, early lock release and group commit have been used as optimization methods in state of the art transaction processing systems. However, in TicToc dealing with this paper, we show that these optimization methods lead to performance degradation when we set the latency of storage to 500 nano second assuming NVRAM with large size of warehouses. When the size of warehouses is relatively small, the phenomenon was not observed.

The reason for the performance degradation with the early lock release was the issuance of log sequence number required by logging. To the best of our knowledge, this is the first case in which both optimization methods, group commit and early lock release, cause performance degradation in a transactional system. We conclude that we need to consider both concurrency control module and recovery module to design an efficient transactional system.

In future work, we try to integrate other state of the art concurrency control protocols with modern recovery protocols including some write ahead logging techniques [17, 22], write behind logging [1], and checkpointing [3]. We also try to use novel NVRAM emulators to improve the quality of the emulation of NVRAM accesses.

Acknowledgment

This work is partially supported by the JST CREST JPMJCR1303 and JPMJCR1414, KAKENHI JP17H01748, and project commissioned by NEDO.

References

[1] Joy Arulraj, Matthew Perron, and Andrew Pavlo. Write-behind logging. *PVLDB*, 10(4):337–348, 2016.

[2] David Blackman and Sebastiano Vigna. xorshift128+. http://xoroshiro.di.unimi.it/xoroshiro128plus.c, 2016.

[3] Nachshon Cohen, David T. Aksun, Hillel Avni, and James R. Larus. Fine-grain checkpointing with in-cache-line logging. 2019.

[4] Jose M. Faleiro, Daniel J. Abadi, and Joseph M. Hellerstein. High performance transactions via early write visibility. *PVLDB*, 10(5):613–624, 2017.

[5] Stavros Harizopoulos, Daniel J. Abadi, Samuel Madden, and Michael Stonebraker. Oltp through the looking glass, and what we found there. In *SIGMOD Conference*, pages 981–992, 2008.

[6] Pat Helland, Harald Sammer, Jim Lyon, Richard Carr, Phil Garrett, and Andreas Reuter. Group commit timers and high volume transaction systems. In *High Performance Transaction Systems*, pages 301–329. Springer, 1989.

[7] Ryan Johnson, Ippokratis Pandis, Radu Stoica, Manos Athanassoulis, and Anastasia Ailamaki. Aether: a scalable approach to logging. *PVLDB*, 3(1-2):681–692, 2010.

[8] Komei Kamiya, Hideyuki Kawashima, Takashi Hoshino, and Osamu Tatebe. Parallel write ahead logging method p-wal. *IPSJ TOD*, 10(1):24–39, 2017.

[9] Hideaki Kimura. Foedus: Oltp engine for a thousand cores and nvram. In *SIGMOD Conference*, pages 691–706, 2015.

[10] Hideaki Kimura, Alkis Simitsis, and Kevin Wilkinson. Janus: Transaction processing of navigation and analytic graph queries on many-core servers. In *CIDR*, 2017.

[11] H. T. Kung and John T. Robinson. On optimistic methods for concurrency control. *ACM TODS*, 6(2):213–226, 1981.

[12] Hyeontaek Lim, Michael Kaminsky, and David G. Andersen. Cicada: Dependably fast multi-core in-memory transactions. In *SIGMOD Conference*, pages 21–35, 2017.

[13] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. Cache craftiness for fast multicore key-value storage. In *EuroSys*, pages 183–196, 2012.

[14] Takayuki Tanabe, Hideyuki Kawashima, and Osamu Tatebe. Integration of parallel write ahead logging and cicada concurrency control method. In *SMARTCOMP*, pages 291–296, 2018.

[15] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. Speedy transactions in multicore in-memory databases. In *SOSP*, pages 18–32, 2013.

[16] Haris Volos, Guilherme Magalhaes, Ludmila Cherkasova, and Jun Li. Quartz: A lightweight performance emulator for persistent memory software. In *Proceedings of the 16th Annual Middleware Conference*, Middleware '15, pages 37–49, New York, NY, USA, 2015. ACM.

[17] Tianzheng Wang and Ryan Johnson. Scalable logging through emerging non-volatile memory. *PVLDB*, 7(10):865–876, 2014.

[18] Tianzheng Wang and Hideaki Kimura. Mostly-optimistic concurrency control for highly contended dynamic workloads on a thousand cores. *PVLDB*, 10(2):49–60, 2016.

[19] Gerhard Weikum and Gottfried Vossen. *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001.

[20] Xiangyao Yu, George Bezerra, Andrew Pavlo, Srinivas Devadas, and Michael Stonebraker. Staring into the abyss: An evaluation of concurrency control with one thousand cores. *Proc. VLDB Endow.*, 8(3):209–220, November 2014.

[21] Xiangyao Yu, Andrew Pavlo, Daniel Sanchez, and Srinivas Devadas. Tictoc: Time traveling optimistic concurrency control. In *SIGMOD Conference*, pages 1629–1642, 2016.

[22] Wenting Zheng, Stephen Tu, Eddie Kohler, and Barbara Liskov. Fast databases with fast durability and recovery through multicore parallelism. In *OSDI*, pages 465–477, 2014.