International Journal of Networking and Computing – www.ijnc.org, ISSN 2185-2847
Volume 14, Number 1, pages 81–92, January 2024

CLMD: Making Lock Manager Predictable and Concurrent for Deterministic Concurrency Control

Masaru Uchida
Faculty of Environment and Information Studies, Keio University
Fujisawa, Kanagawa, 252-0882, Japan

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

Received: July 30, 2023
Accepted: September 8, 2023
Communicated by Susumu Matsumae

Abstract

Transaction processing is common in our daily lives, and various protocols have been studied to manage concurrency control. One such protocol is the deterministic concurrency control protocol, which is highly efficient in handling workloads with high contention because of its deterministic transaction scheduling nature. This paper introduces a new lock manager, the concurrent lock manager for determinism or CLMD. CLMD allows non-conflicting transactions to be executed concurrently on deterministic concurrency control protocols. The concept is to check for conflicts between the current and all future transactions while the conventional scheme checks only the current and the following ones. The CLMD eliminates the bottleneck present in the conventional locking scheme used in the original Calvin paper. We evaluated CLMD with Calvin in experiments. The results showed that the proposed method outperformed SS2PL under high-contention workloads and performed better than Calvin under low and high-contention workloads. The maximum performance improvement observed was approximately 44.4 times using 64 logical cores.

Keywords: Transaction, Concurrency Control, Determinism

1 Introduction

1.1 Motivation

Transaction processing is widely used today. Its application includes payments [23], distributed file systems [9], and robotics [22] today. To deal with huge amounts of transactions, various studies have been conducted in these years [4, 12, 14, 16–18, 24, 25, 28, 31].

When multiple transactions access the same records simultaneously, only one transaction survives, and all the others must wait or abort. When the degree of concurrently accessing transactions is high, the situation is highly contended. Some workloads, such as YCSB [3] or TPC-C [26], use relatively short transactions. YCSB has only 16 operations and the majority of TPC-C transactions;

[0]The conference version of this paper is published in the proceedings of 25th Workshop on Advances in Parallel and Distributed Computational Models (APDCM 2023).

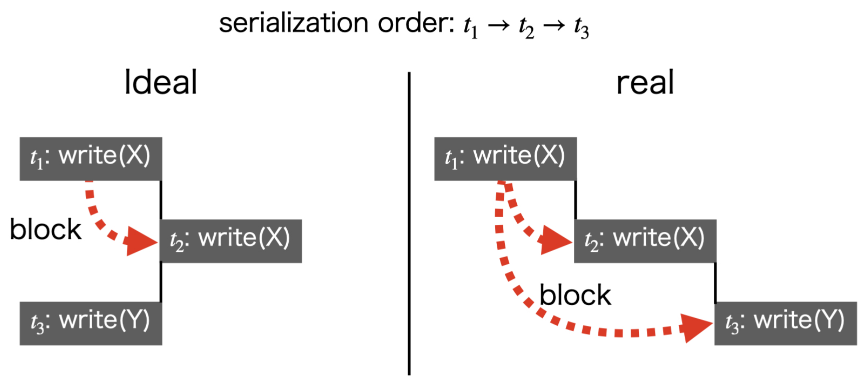

Figure 1: Three transactions run concurrently and the serialization order is $t_1 \rightarrow t_2 \rightarrow t_3$. An unnecessary block of t_3 occurs when using the Calvin protocol as shown in the right figure. Though t_3 does not conflict with either t_1 or t_2, it should not be blocked in theory. Since t_3 is scheduled after $t2$ and $t2$ is blocked by $t1$, t_3 is blocked.

the new order and payment do not require range queries. However, some real workloads require long transactions. The bill of materials workload requires long transactions [19], and transactional analytics for social data requires range queries of over 50,000 records [1].

1.2 Problem

To deal with highly contended situations in concurrency control, various protocols have been studied [12,14,17,19,29,30]. 2 phase locking (2PL) based protocols [2,17,20,21,27] pessimistically access records to reduce aborts. Multi-version concurrency control (MVCC) based protocols [11,12,14,19] exploit the theoretically larger scheduling space than 2PL.

One of the promising approaches is *determinism* [29], which determines the execution order of a set of transactions before they are executed, and its novelty for highly contended workloads than 2PL and MVCC is already investigated in a literature [8]. The theory of determinism is that conflicting transactions are executed sequentially one by one, and independent ones are executed in parallel. The transaction executions are controlled by a *lock manager* (e.g. pthread_mutex_lock, a function that waits until the lock is released and can be acquired when it is already locked).

1.3 Contribution

The reason for the problem illustrated in Figure 1 is that t_3 waits unnecessarily. Since t_3 is scheduled after t_2 and t_2 is blocked by a lock manager due to a conflict with t_1, t_3 is not invoked. We can improve the throughput if we can execute t_1 and t_3 simultaneously.

To solve this problem, we propose a novel lock manager architecture that checks the conflicts between the waiting transactions and running but blocked transactions. We refer to our proposed system as the *concurrent lock manager for determinism* or `CLMD`. The behavior of `CLMD` differs from that of the conventional lock manager. When a transaction requests a lock from `CLMD`, the request is registered with `CLMD` and the transaction starts waiting instead of blocking. When the lock request is ready, the transaction is notified and acquires the lock. Therefore, all lock requests from all transactions are registered with `CLMD` in the first phase. Then theoretically, concurrent executable transactions are executed in parallel in the second phase. We designed and implemented `CLMD` and evaluated its performance. The results showed that Calvin with `CLMD` performed approximately 44.4 times higher throughput than that of Calvin with pthread_mutex_lock.

1.4 Organization

The rest of this paper is organized as follows: Section 2 describes Calvin, a representative deterministic concurrency control protocol; Section 3 proposes a novel concurrent lock manager for deterministic databases; Section 4 evaluates the proposed method; Section 5 describes related work; finally, Section 6 concludes this paper.

2 Preliminaries

2.1 Concurrency Control

Transaction processing is necessary for the correct concurrent execution of processes. The transaction processing guarantees the ACID properties [33], and it has two modules. They are the concurrency control module and the logging and recovery module. This paper focuses only on the concurrency control module like some other studies [12, 14, 16, 17, 27, 28, 34]. The concurrency control is for providing isolation property, and the strongest level is called serializable. This paper focuses only on the serializable isolation level.

A variety of concurrency control protocols have been studied to provide serializability and high performance over concurrent execution transactions. The protocols can be divided into two categories: deterministic [5–7, 15, 25, 30] or non-deterministic [28]. The former assumes all operations in all transactions are known before their executions, while the latter does not have such an assumption. It should be noted that the former situation is not universal but is possible in real use cases such as payments or bank transfers at ATMs [23].

2.2 Database Operations and Architecture

The database system provides operations for users to interact with the database. In this paper, we focus on read and write as these operations. The read operation reads a record, and the write operation updates a record in the database, respectively.

An operation can access records in the database via a variety of protocols. A single transaction has multiple operations (i.e., read or write), and a worker thread executes transactions one by one. A worker thread is assigned to a CPU core in this paper. Thus, for N logical CPU cores, N worker threads run. The database architecture is illustrated in Figure 2.

2.3 Calvin: A Deterministic Protocol

Calvin [30] is a representative of the deterministic protocols. A worker thread under Calvin has two phases to operate the set of transactions. In the first phase, the *sequencing layer* determines a serialization order of the transactions.

Then, in the second phase, the *scheduling layer* controls a transaction t_i to perform as summarized in Table 1. It has four steps. It first acquires the giant lock and then tries to acquire record locks for t_i. If t_i does not encounter any conflicts in this step, another transaction t_j will perform the first step. Otherwise, t_i stops its execution, and subsequent transactions are blocked by it. If the conventional lock manager is used for Calvin, an inappropriate block occurs in the second step, which is illustrated in Figure 1. Such blocks should be avoided for providing high concurrency. The details of Calvin's protocol are illustrated in Alg. 1.

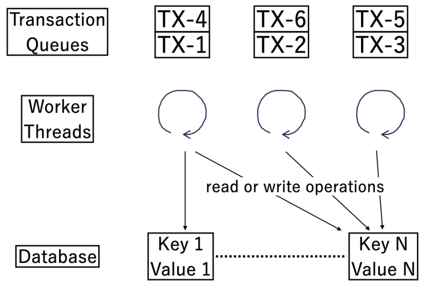

Figure 2: Database Architecture. A worker thread executes transactions one by one. A transaction includes multiple operations (read or write). An operation accesses a record which consists of a key and a value.

Table 1: Overview of Calvin Locking Protocol (extracted from Alg. 1.)

1.	Acquire the giant lock (L7)
2.	Acquire locks for t_i, it may be blocked (L8)
3.	Unlock the giant lock (L16)
4.	Do operations and release locks (L19)

3 Proposal: Concurrent Lock Manager for Determinism

To solve the blocking problem that occurs in the conventional lock manager, we propose a novel *concurrent lock manager for determinism* (CLMD). The proposed CLMD delays the lock acquisition timing at the *scheduling layer* for providing high concurrency. After a lock acquisition request is registered with CLMD, the requested transaction waits until the lock is ready. This scheme allows multiple non-conflicting transactions to acquire locks concurrently.

A running example of CLMD is illustrated in Figure 3. Four transactions are already ordered as $t1$, $t2$, $t3$, and $t4$ in the *sequencing layer.* The transactions are operated by workers A, B, C, and D, respectively. Since $t3$ does not conflict with either $t1$ or $t2$, CLMD provides locks for $t1$ and $t3$ simultaneously. However, $t3$ is blocked if the conventional lock manager is used.

The behavior of CLMD is summarized in Table 2. It first acquires the giant lock as the conventional lock manager, as in Table 1. Then it just registers lock requests for the desired records and immediately unlocks the giant lock. Thus, it does not incur unnecessary blocks of subsequent transactions. When the transaction is ready to use the registered lock requests, then it is notified by CLMD.

Using CLMD, a transaction t_i in the *scheduling layer* works as above, and the details are illustrated in Alg. 2. All records to be accessed are stored in the operation set O. At the beginning of a transaction, *tx_id* is added to the queue of all records to be accessed. Since this process is exclusive, the order in which *tx_id* is stored in the queue follows the serialization order. The transaction needs to know which records are not yet locked, so information on the records to which *tx_id* is added is

Table 2: Modified Calvin with CLMD (from Alg. 2.)

1.	Acquire the giant lock (L8)
2.	Register lock requests for t_i **without blocking** (L11)
3.	Unlock the giant lock (L14)
4.	Acquire locks for t_i (L21)
5.	Do operations and release locks (L3)

Algorithm 1 Worker behavior in Calvin

```
Require: A set of operations $O$ for a transaction. An operation is either to read a record or write a record.
1: function WORKER_CALVIN($O$)
2:     Calvin_Locking($O$);
3:     Exec_Operation($O$);
4: end function
5:
6: function CALVIN_LOCKING($O$)
7:     Acquire the giant lock;
8:     for $opr \in O$ do
9:         if target record $rec$ for $opr$ is unlocked then
10:            Acquire lock of $rec$;
11:        else
12:            Blocked until lock of $rec$ is released;
13:            Acquire lock of $rec$;
14:        end if
15:    end for
16:    Release the giant lock;
17: end function
18:
19: function EXEC_OPERATIONS($O$)
20:    for $opr \in O$ do
21:        Execute $opr$ to its target record;
22:    end for
23:    for $opr \in O$ do
24:        Release the lock of $rec$ for $opr$;
25:    end for
26: end function
```

recorded on the future lock set F. When tx_id is added to the queue of all records to be accessed, the giant lock is released, and exclusive control is terminated. CLMD does not block transactions even if they are conflicted with previous ones since the actual lock acquisitions occur later.

The future lock set F contains information on the records that need to be locked. A transaction checks the queue of records in F. If tx_id, which was exclusively added at the beginning of the transaction, is at the head of the queue, the transaction can try to acquire the lock on the record because no other transaction has already requested lock acquisition before it. If the lock acquisition succeeds, it deletes its own tx_id from the queue of the record and removes the record from F for which the lock was acquired. If the first tx_id in the queue is not its own tx_id or it fails to acquire the lock, it tries to acquire a lock on another record in F. When all requested locks are acquired and F is empty, it starts to execute read or write operations.

4 Evaluation

4.1 Implementation and Environment

We developed CLMD and Calvin to evaluate our proposed method. The system was developed using C++ on CCBench [28]. We compare the proposed system with strong strict 2-phase locking (SS2PL) and Calvin with pthread_mutex_lock. After all threads were launched, time measurements were started, and throughput was calculated as a 3-second average. In all methods, the type of operations and the records to be accessed, which are the contents of the transaction, are prepared in advance. Threads retrieve the transactions one by one from the previously prepared transaction set and execute them one by one. Our code is over 600 lines long and available online for reproduction [32].

We used a server with two sockets equipped with Intel(R) Xeon(R) Gold 6130 CPU @ 2.10GHz for the experiments. Each socket had 16 physical and 32 logical cores. The total size of memory was 1.5 TB. The number of operations performed by each transaction was 10 read or write operations

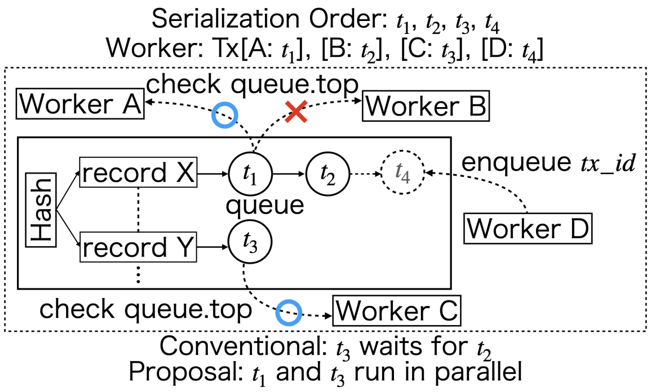
Serialization Order: t1, t2, t3, t4
Worker: Tx[A: t1], [B: t2], [C: t3], [D: t4]
check queue.top
Worker A
Worker B
Hash
record X
record Y
t1
t2
t4
t3
queue
enqueue tx_id
Worker D
check queue.top
Worker C
Conventional: t3 waits for t2
Proposal: t1 and t3 run in parallel

Figure 3: `CLMD` Architecture. Worker threads A, B, C, and D perform $t1$, $t2$, $t3$, and $t4$, respectively. $t1$, $t2$, and $t4$ access record X while $t3$ accesses only Y. Access is via a hash. $t1$, $t2$, and $t4$ are queued for X. The conventional lock manager requires $t3$ to stop by the end of $t2$ locking, while `CLMD` does not.

with a probability of 50% each and one sleep operation of 100 μs which was the ninth operation of the total. For the first two transactions, we always access the conflicting record and perform a sleep of 2.9 seconds. The database has 1,000,000 records. In the workload shown in Figure 7 and 8, no specific transaction accesses a specific record, and all transactions consist of 100 read operations or write operations with a probability of 50% each, and one sleep of 1000 μs, which is the 90-th operation of the entire transaction.

4.2 Result with Long Transactions

4.2.1 Varying Skew

The throughput under the relatively low contention workload with long transactions is illustrated in Figure 4. In this experiment, we varied the skew from 0.0 to 0.9. It should be noted that when the skew is 0.0, a transaction can access all the database records. With a higher skew value, the accessible records decrease and conflicts tend to occur. The maximum skew is 1.0.

The result shows that the proposed method always outperforms the conventional method regardless of the frequency of contention. Both the proposed and conventional methods perform best when the skew is 0.0, and the performance degrades as the frequency of contention increases. Even when the performance difference between the two methods is the smallest, the difference is 2.2 times larger, and when the performance difference is the largest, the difference is 44.4 times larger.

In the proposed method, non-conflicting transactions can acquire locks in parallel, but the conventional method acquires locks sequentially. In this environment, the conventional method sleeps for 2.9s while the first transaction holds the lock on record X. The second transaction also accesses X, thus, it makes a lock request on X. The second transaction's lock request on X can be obtained, and the third and subsequent transactions can request locks on all records only after the first transaction sleeps for 2.9s and releases its lock on X.

The conflict between the first and second transactions causes serious performance degradation because the third and subsequent transactions cannot start processing even though they are not in conflict.

4.2.2 Scalability

Figure 5 shows the result of the experiment with thread scalability. It is observed that throughput increases as the number of threads increases for both the conventional and proposed methods. The proposed method always performs better than the conventional method, and the difference in

Algorithm 2 Worker behavior in proposal method

```
Require: A set of operations $O$ for a transaction. An operation is either to read a record or write a record.
    Future lock set $F$. Transaction ID $tx\_id$
function WORKER_PROPOSED($O, F, tx\_id$)
    Proposed_Locking($O, F, tx\_id$);
    Exec_Operation($O$);                                   ▷ Described in Alg. 1
end function

function PROPOSED_LOCKING($O, F, tx\_id$)
    // Lock Request Registration
    Acquire the giant lock;
    for $opr \in O$ do
        $rec$ := target of $opr$;
        Enqueue $tx\_id$ into queue of $rec$;
        Add $rec$ to $F$;
    end for
    Release the giant lock;

    // Lock Acquisition
    while $F$ is not $\emptyset$ do
        for $rec \in F$ do
            if $rec$.queue.head is $tx\_id$ then
                if $rec$ is unlocked then
                    Acquire lock of $rec$;
                    Dequeue $tx\_id$ from $rec$.queue;
                    Delete $rec$ from $F$;
                end if
            end if
        end for
    end while
end function
```

throughput increases as the number of threads increases. The reason for the low scalability of the conventional method is the blocking by the first transaction. Since most of the threads are blocked by a single thread that executes the first transaction, which sleeps for 2.9 seconds, the many-core architecture is not exploited.

4.2.3 Varying Execution Time of Blocking Transaction

To investigate the behavior of the proposed method, we varied the execution time of the first transaction. The first transaction always blocks the second transaction, and thus using the conventional lock manager, subsequent transactions are blocked until the first transaction terminates.

The result of the experiment is illustrated in Figure 6. In the conventional method, the conflict between the first two transactions causes the first transaction to terminate without allowing the third and subsequent transactions to start executing, thus only one transaction succeeds during the experiment time, regardless of the execution time of the first transaction. In the proposed method, the longer the execution time of the first transaction is extended, the more non-conflicting transactions that can be executed in the meantime are executed. Therefore, the number of successful transactions increases linearly with the execution time of the first transaction.

This setting has only a single blocking transaction, and the performance difference could be greater if we have more blocking transactions in the workload.

4.3 Result without Long Transactions

The proposed method should work efficiently with long transactions that block other transactions. Then, does it work for workloads without such long transactions? To answer the question, we

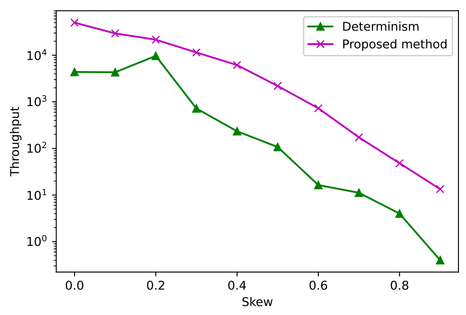

Figure 4: Low-contention workload with long transactions.

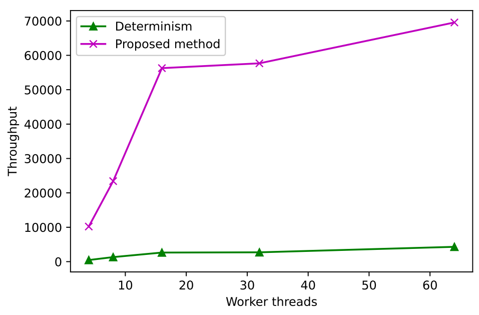

Figure 5: Scalability in low-contention workload with long transactions.

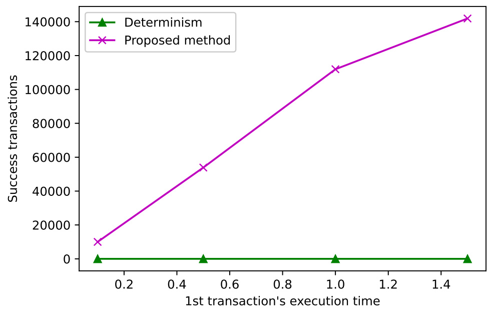

Figure 6: #transactions to be executed while the first transaction sleeps.

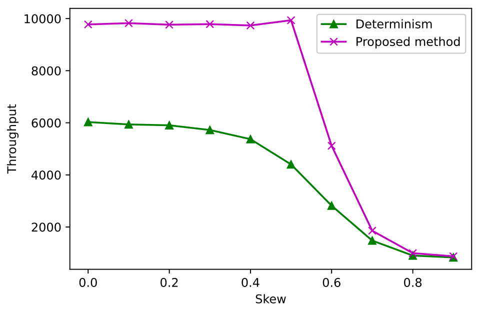

Figure 7: Low-contention workload without long transactions.

conducted experiments.

4.3.1 Low Contention Case

Figure 7 shows the result when all transactions are executed without long transactions, which means no special sleep exists. Although the difference in performance between the proposed method and the conventional method is not as large as in the other experiments, the proposed method still shows higher performance because there are cases where a transaction with no conflict is forced to wait due to a conflict between transactions ahead of it in the conventional method.

In this experiment, the difference in performance between the conventional and proposed methods shrinks as the skew increases. One possible reason for this is that transactions are executed like sequential execution; in the high-contention workload, almost all transactions conflict with each other. In such cases, Calvin executes transactions sequentially. Then, the proposed method is more likely to cause performance degradation since it is more complex than the conventional method.

4.3.2 High Contention Case

We also evaluated the proposed method in a highly contended case with SS2PL. The result comparing the proposed method and SS2PL on high-contention workload is shown in Figure 8. Our SS2PL uses the 2PL-NoWait [28] method, which aborts and restarts the transaction if it fails to acquire a lock for deadlock avoidance. Therefore, when conflicts occur frequently, lock acquisition fails, and the transactions repeatedly abort. Therefore, the performance of SS2PL is lower than that of the conventional and proposed methods that are abort-free and ensure sequential execution.

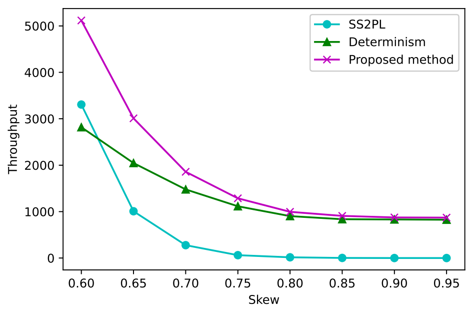

Figure 8: High-contention workload without long transactions.

5 Related Work

Piece-wise-visibility (PWV) [6] is a deterministic protocol that improves processing performance by accelerating the timing at which a transaction write can be observed by other transactions, taking advantage of the characteristics of deterministic methods such as deadlock handling and the absence of system-induced aborts such as failures and optimistic concurrency control validation failures. In PWV, a transaction is decomposed into multiple pieces to deal with aborts caused by logic (e.g., trying to reduce inventory by purchasing an item that was initially zero). Records in the database are divided into multiple exclusive partitions, and each piece writes only to records in one partition. The serializability is guaranteed by executing different transaction pieces within a partition according to the transaction serialization order. Specifically, partitions understand the conflicts between pieces of different transactions and construct a DAG that indicates the dependencies of the pieces according to the determined serialization order. It executes the executable pieces according to the directed acyclic graph (DAG). The construction of such a DAG not only occupies the resources of the CPU cores, of which only one is allocated to each partition, and prevents the processing of the pieces that are the actual contents of the transaction but also requires an enormous amount of memory to maintain the DAG. Our proposed method does not require excessive CPU and memory resources and can increase the parallelism of the execution of independent transactions.

Calvin [30] shows its true value in a distributed environment where data is partitioned across multiple nodes and replicated to other machines on a partition-by-partition basis since it proposes concurrency control and replication. Its concurrency control is a deterministic method that controls the results of transactions executed in parallel to match those executed sequentially in a specific order. By taking advantage of this property, the transaction inputs and locking orders are shared at the start of batch processing via a consensus protocol such as Paxos [13] or Zookeeper [10], and the different nodes are deterministically attributed to a certain outcome, thus eliminating the overhead of performing mediation such as two-phase commit after the transaction starts, which is used in general distributed systems. In Calvin, each node has a lock manager that manages only the locks on the records in that node. It does not exchange locks with other nodes. Therefore, we have implemented and evaluated our proposed concurrent lock manager on a single node, and we believe that even if Calvin were implemented as a multi-node system and our concurrent lock manager were evaluated, the lock manager would still work the same way and perform better than conventional lock managers.

Aria [15] executes transactions against database snapshots and deterministically chooses whether to commit at transaction commit time. This removes the constraints of determinism, where all database operations must be known in advance, and allows the database system to adapt to a wide range of workloads. Aria commits only those transactions that access records in a predetermined order at commit time and aborts and re-executes those that do not until a deterministic result is reached. In such a method, access to a particular record may be concentrated and the order in which transactions access that record violates the deterministically determined order, and many transactions abort and retry at times of high contention. In such cases, the performance of deterministic methods may be severely degraded without taking advantage of their high-competition performance against non-deterministic methods such as 2PL and OCC under high-contention workloads. In the case where the key of the record to be accessed is unknown at the start of the transaction, which is the case where Aria is expected to show its true value, conventional deterministic methods such as Calvin can be used by using a reconnaissance query method to find out the access key in advance.

6 Conclusions

Deterministic concurrency control protocols exhibit high performance at highly contended workloads. This paper proposes a novel lock manager that allows non-conflicting transactions to execute concurrently on deterministic concurrency control protocols. The concept is to check conflicts between the current and the next transactions and between the current and all future transactions. The proposed lock manager eliminates the bottleneck of the conventional locking schema used in Calvin [30]. We conducted experiments to compare it with the widely used SS2PL and Calvin with the conventional lock manager. The results showed that the proposed method outperformed SS2PL under high-contention workload and showed higher performance than conventional Calvin under both low and high-contended workloads. The maximum performance improvement is approximately 44.4 times using 64 logical cores.

Acknowledgement

This paper is based on results obtained from "Research and Development Project of the Enhanced Infrastructures for Post-5G Information and Communication Systems" (JPNP20017), commissioned by the New Energy and Industrial Technology Development Organization (NEDO) and JSPS KAKENHI Grant Number 22H03596.

References

[1] J. Chen, Y. Ding, Y. Liu, F. Li, L. Zhang, M. Zhang, K. Wei, L. Cao, D. Zou, Y. Liu, L. Zhang, R. Shi, W. Ding, K. Wu, S. Luo, J. Sun, and Y. Liang. Bytehtap: Bytedance's htap system with high data freshness and strong data consistency. *PVLDB*, 15(12):3411–3424, 2022.

[2] Y. Chen, X. Yu, P. Koutris, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and J. Shu. Plor: General transactions with predictable, low tail latency. In *SIGMOD Conf.*, pages 19–33, 2022.

[3] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *SoCC*, pages 143–154. ACM, 2010.

[4] K. Doki, T. Hoshino, and H. Kawashima. Accelerating scan transaction with node locking. In *RTCSA*, pages 101–106, 2023.

[5] J. M. Faleiro and D. J. Abadi. Rethinking serializable multiversion concurrency control. *PVLDB*, 8(11):1190–1201, 2015.

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

[7] H. Fan and W. Golab. Ocean vista: Gossip-based visibility control for speedy geo-distributed transactions. *PVLDB*, 12(11):1471–1484, 2019.

[8] R. Harding, D. V. Aken, A. Pavlo, and M. Stonebraker. An evaluation of distributed concurrency control. *PVLDB*, 10(5):553–564, 2017.

[9] K. Hiraga, O. Tatebe, and H. Kawashima. Scalable distributed metadata server based on nonblocking transactions. *J. Univers. Comput. Sci.*, 26(1):89–106, 2020.

[10] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *USENIXATC'10*, page 11, 2010.

[11] T. Kambayashi, T. Tanabe, T. Hoshino, and H. Kawashima. Shirakami: A hybrid concurrency control protocol for tsurugi relational database system. *CoRR*, abs/2303.18142, 2023.

[12] K. Kim, T. Wang, R. Johnson, and I. Pandis. Ermia: Fast memory-optimized database system for heterogeneous workloads. In *SIGMOD Conf.*, page 1675–1687, 2016.

[13] L. Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16:133–169, may 1998.

[14] H. Lim, M. Kaminsky, and D. G. Andersen. Cicada: Dependably fast multi-core in-memory transactions. In *SIGMOD Conf.*, page 21–35, 2017.

[15] Y. Lu, X. Yu, L. Cao, and S. Madden. Aria: A fast and practical deterministic oltp database. *PVLDB*, 13(12):2047–2060, 2020.

[16] K. Masumura, T. Hoshino, and H. Kawashima. Fast concurrency control with thread activity management beyond backoff. *J. Inf. Process.*, 30:552–561, 2022.

[17] T. Nakamori, J. Nemoto, T. Hoshino, and H. Kawashima. Removing performance bottleneck of timestamp allocation in two-phase locking based protocol. In *ADC*, page 201–208, 2022.

[18] S. Nakazono, H. Uchiyama, Y. Fujiwara, and H. Kawashima. Scheduling space expander: An extension of concurrency control for data ingestion queries. *CoRR*, abs/2301.10440, 2023.

[19] J. Nemoto, T. Kambayashi, T. Hoshino, and H. Kawashima. Oze: Decentralized graph-based concurrency control for real-world long transactions on bom benchmark, 2022.

[20] T. Nguyen and H. Kawashima. Fairly decentralizing a hybrid concurrency control protocol for real-time database systems. In *CANDAR CSA*, 2023.

[21] T. Nguyen and H. Kawashima. Make PLOR real-time and fairly decentralized. In *RTCSA*, pages 273–274, 2023.

[22] Y. Ogiwara, A. Yorozu, A. Ohya, and H. Kawashima. Transactional transform library for ros. In *IROS*, pages 9722–9727, 2022.

[23] T. Ogura, Y. Akita, Y. Miyazawa, and H. Kawashima. Accelerating geo-distributed transaction processing with fast logging. In *IEEE BigData*, pages 2390–2399. IEEE, 2021.

[24] R. Onishi, T. Hoshino, and H. Kawashima. EPO-R: an efficient garbage collection scheme for long-term transactions. In *CANDAR*, pages 144–150, 2022.

[25] D. Qin, A. D. Brown, and A. Goel. Caracal: Contention management with deterministic concurrency control. In *SOSP*, page 180–194, 2021.

[26] F. Raab. Tpc-c - the standard benchmark for online transaction processing (oltp). In J. Gray, editor, *The Benchmark Handbook*. Morgan Kaufmann, 1993.

[27] P. Ramalhete, A. Correia, and P. Felber. 2plsf: Two-phase locking with starvation-freedom. In *PPoPP*, pages 39–51, 2023.

[28] T. Tanabe, T. Hoshino, H. Kawashima, and O. Tatebe. An analysis of concurrency control protocols for in-memory databases with ccbench. *PVLDB*, 13(13):3531–3544, 2020.

[29] A. Thomson and D. J. Abadi. The case for determinism in database systems. *PVLDB*, 3(1–2):70–80, 2010.

[30] A. Thomson, T. Diamond, S.-C. Weng, K. Ren, P. Shao, and D. J. Abadi. Calvin: Fast distributed transactions for partitioned database systems. In *SIGMOD Conf.*, page 1–12, 2012.

[31] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden. Speedy transactions in multicore in-memory databases. In *SOSP*, page 18–32, 2013.

[32] M. Uchida. Code of concurrent lock manager for deterministic protocols on CCBench. `https://github.com/masa1u/ccbench/tree/master/lock-manager`.

[33] G. Weikum and G. Vossen. *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery.* Morgan Kaufmann, 2002.

[34] X. Yu, A. Pavlo, D. Sanchez, and S. Devadas. Tictoc: Time traveling optimistic concurrency control. In *SIGMOD Conf.*, page 1629–1642, 2016.