TSINGHUA SCIENCE AND TECHNOLOGY
ISSN 1007-0214 05/10 pp925–938
DOI: 10.26599/TST.2021.9010036
Volume 27, Number 6, December 2022

Multi-Clock Snapshot Isolation Concurrency Control for NVM Database

Xuyang Liu, Kang Chen*, Mengxing Liu, Shiyu Cai, Yongwei Wu, and Weimin Zheng

Abstract: Multi-Clock Snapshot Isolation (MCSI) is a concurrency control mechanism that implements snapshot isolation on a single-layer Non-Volatile Memory (NVM) database. It stores a single copy of data by using multi-version storage to ensure durability and runtime access. With multi-clock transaction timestamp assignment, MCSI can efficiently generate snapshots with vector clocks and use per-thread transaction status arrays to identify uncommitted versions in NVM. For evaluation, we compared MCSI with the PostgreSQL-style concurrency control used in the single-layer NVM database N2DB. The maximum transaction throughput of MCSI is 101%–195% higher than that of N2DB for the YCSB workloads, and 25%–49% higher for the TPC-C workloads. Moreover, the transaction latency of MCSI remains relatively stable as the thread count increases. With 18 worker threads, the average transaction latency of MCSI is 65%–84% lower than that of N2DB for the YCSB workloads and 16%–43% lower for the TPC-C workloads.

Key words: Non-Volatile Memory (NVM); snapshot isolation; Multi-Version Concurrency Control (MVCC); vector clock

1 Introduction

Traditional DataBase Management Systems (DBMSs) use a two-layer storage architecture. One layer is the volatile, fast, byte-addressable main memory (DRAM), and the other layer is the non-volatile, relatively slow, block-addressed disk (HDD or SSD). Disk-oriented DBMSs store all data on large but slow disks, and they use buffer pools in the main memory to speed up data access, while in-memory DBMSs store all the data in the main memory and use disks to ensure durability. In-memory DBMSs maintain two copies of data: in-memory data for faster runtime access and on-disk data for durability.

Emerging Non-Volatile Memory (NVM) combines the properties of the main memory and disks. It is byte-addressable, persistent, and has a similar access latency as DRAM. It has been used to replace components in traditional database systems, such as on-disk file systems[1–3], buffer pool memory[4, 5], checkpoint, log storage[6–8], and index[9–11].

However, current works still rely on the traditional two-layer database architecture, which is arguably not ideal for NVM[4]. A more promising approach is to abandon the traditional storage architecture and leverage NVM properties to build single-layer databases, which maintain only a single copy of the data for both durability and runtime access. Multi-Version Concurrency Control (MVCC) is a natural choice for such databases[12–14], because the multi-version storage in MVCC allows higher concurrency at runtime and is good for crash recovery in NVM.

Concurrency control describes how transactions are executed and coordinated to ensure certain guarantees, such as atomicity, consistency, isolation, and durability.

• Xuyang Liu, Kang Chen, Mengxing Liu, Shiyu Cai, Yongwei Wu, and Weimin Zheng are with Department of Computer Science and Technology, Tsinghua University, Beijing 100084, China. E-mail: liuxuyan18@mails.tsinghua.edu.cn; chenkang@tsinghua.edu.cn; liu-mx15@mails.tsinghua.edu.cn; caisy18@mails.tsinghua.edu.cn; wuyw@tsinghua.edu.cn; zwm-dcs@tsinghua.edu.cn.

* To whom correspondence should be addressed.

Manuscript received: 2021-02-03; revised: 2021-03-06; accepted: 2021-05-06

Isolation level defines the degree to which a transaction is isolated from changes made by other transactions. Snapshot isolation[15] is an isolation level in which each transaction sees a consistent snapshot of the database during its execution regardless of changes made by other concurrent transactions. Snapshot isolation is implemented in many real-world MVCC database systems[16–18]. Supporting snapshot isolation in MVCC databases is worthwhile for two reasons. First, with snapshot isolation, transactions can leverage MVCC versions in the snapshot to improve concurrency. Second, when a stronger isolation level is needed, the database can use serializable certifiers[19, 20] on top of snapshot isolation to reach serializability. Therefore, efficiently implementing snapshot isolation plays an important role when building single-layer NVM databases with MVCC.

The concurrency control implementations from traditional database systems may experience performance problems when adapted to NVM either because they focus on the in-memory data and do not take persistence overheads into account or because faster NVMs exaggerate the scalability issues in their designs. For example, a previous single-layer NVM database N2DB[14] used the snapshot isolation from PostgreSQL[17]. Its snapshot generation process involves taking a lock and scanning a global active transaction list, which blocks concurrent transactions and is not scalable. Also, N2DB adopted PostgreSQL's commit log for durability and crash consistency. However, such a design experiences the false-sharing problem in NVM.

To solve these issues, we present Multi-Clock Snapshot Isolation (MCSI), which efficiently implements snapshot isolation in a single-layer NVM database. MCSI stores a single copy of data in NVM for both runtime access and durability. It handles crash consistency by storing transactions' commit status in NVM to identify uncommitted changes after a crash. Like in PostgreSQL, MCSI represents a snapshot through all currently completed transactions in the system. Each worker thread has its own local clock for assigning transaction timestamps; thus, MCSI can asynchronously scan the timestamp of the last completed transaction from all threads to generate snapshots, and each worker thread can use its own single-writer status array in NVM to store the transaction status instead of a shared commit log. Using those techniques, MCSI can provide high throughput and low-latency transaction processing. MCSI scales well and can maintain the low transaction latency as the thread number increases. Experiments using the YCSB and TPC-C benchmarks on real hardware show that MCSI can achieve 25% to 195% higher transaction throughput and 16% to 84% lower transaction latency compared with the PostgreSQL-style snapshot isolation implementation in N2DB.

The remainder of this paper is organized as follows. Section 2 introduces the background. Section 3 describes MCSI's single-layer database storage in NVM. Section 4 discusses how we use multi-clock to implement snapshot isolation. Section 5 evaluates MCSI against the PostgreSQL-style snapshot isolation in N2DB. Section 6 presents the related work. Section 7 concludes this paper.

2 Background

2.1 Non-volatile memory

Traditional databases are built on systems with fast, volatile DRAMs and slow, non-volatile disks. With the nature of the two-layer storage architecture, traditional databases must store two copies of their data—one in DRAM for fast access and one on the disk for durability.

Emerging NVM offers a promising blend of the two types of storage devices. NVM devices are low-latency, byte-addressable, and can be plugged in DIMM slots. They are considered parts of the main memory and are directly accessible by the CPU using load or store instructions. They are also non-volatile, which makes them a good replacement for disks to store persistent data. A comparison of DRAM, Intel Optane DC NVMs, and SSD is shown in Table 1.

Data written by using store instructions to NVM are not guaranteed to be persistent due to the modern CPU cache hierarchy. Data will be first buffered in the CPU cache and kept volatile until they are flushed to NVM either by an implicit cache eviction policy (not visible to programmers) or explicit cache flush instructions such

Table 1 Characteristics of DRAM, NVM, and SSD.

	Read latency	Write latency	Capacity	Bandwidth	Byte-addressable	Non-volatile
DRAM	80–100 ns	57 ns	128 GB	~10 GB/s	√	×
Optane DC NVM	170–300 ns	62 ns	512 GB	2–6 GB/s	√	√
SSD	100 μs	14 μs	Several TB	~2 GB/s	×	√

as CLFLUSH, CLFLUSHOPT, and CLWB. CLWB is preferred if available because it does not invalidate the cache line. Other instructions are also applicable for storing data in NVM. NTSTORE can bypass the cache hierarchy and directly write to NVM, which is more efficient for large writes[21]. Programmers must invoke these instructions at appropriate times to ensure crash consistency of their persistent data structures in NVM.

2.2 MVCC and snapshot isolation

MVCC has become the most popular transaction concurrent control scheme for recent high-performance database systems[22–25]. In MVCC, transactions always create new physical versions of tuples (records) instead of updating them in place directly when making changes to the database. The database maintains multi-version storage for tuples to keep both the new and old versions[26]. MVCC databases typically implement snapshot isolation, in which every transaction sees a consistent snapshot of the database taken before its start time.

MVCC has also been used to build single-layer databases in NVM[13, 14]. The use of MVCC in NVM has two advantages. (1) MVCC allows more concurrency because while a record is being updated, its before-image is always available, so concurrent readers may not be blocked. With MVCC and snapshot isolation, read-only transactions can make progress regardless of concurrent writers. This feature is beneficial for fast NVM and DRAM devices. (2) The second reason is related to crash recovery. In a single-layer NVM database, transactions will directly modify the data stored in NVM. If the system crashes while a transaction is still running, then we need to recover the data to a consistent state before the transaction makes any modification. With multi-version storage, we can discard the dirty changes from interrupted transactions and read the previously committed version. An additional recovery process like ARIES is not needed[27].

2.3 Concurrency control for NVM databases

Concurrency control describes how the transactions are executed and coordinated. Isolation level defines the degree to which a transaction is isolated from changes made by other transactions. Efficient implementation of snapshot isolation plays an important role when building single-layer NVM databases with MVCC. The snapshot isolation concurrency control implementation used in traditional MVCC databases is not well suited for NVM because of two reasons. (1) The concurrency control implementations of traditional databases mainly focus on their in-memory part without discussing durability and crash consistency. They leave those to another dedicated recovery protocol, such as ARIES or write-ahead logging. The situation is different in NVM databases because a single copy of data is used for both runtime access and durability. Ensuring crash consistency should unavoidably be part of the concurrency control itself. (2) Concurrency control methods from a two-layer architecture have suboptimal performance when directly applied to a single-layer NVM database. FOEDUS[28] points out that centralized components, such as lock manager and logs in traditional concurrency control implementations, will cause performance bottlenecks in a multicore NVM database. A previous NVM database N2DB[14] adopted the concurrency control from PostgreSQL[17] to implement snapshot isolation. The concurrency control scheme from PostgreSQL uses the currently completed transactions in the system to represent a snapshot. When generating a snapshot, it has to take a lock and scan the global active transaction list. This approach blocks concurrent transactions and is not scalable. N2DB also adopted the commit log in NVM to persistently store transaction status for crash consistency. The commit log is an array-like structure and is write-shared by all threads, which limits its scalability. Zen[29] also points out that traditional concurrency control methods often need to modify the tuple metadata not only by tuple writes but also by tuple reads, which incurs expensive NVM writes.

3 Single-Layer Database Storage in NVM

3.1 Overview

The architecture of MCSI's single-layer storage in NVM is depicted in Fig. 1. NVM stores two types of data for the database: tables and transaction status. Tables maintain the multi-version records (tuples) in NVM. The per-thread transaction status array in NVM stores the status (initial, running, committed, and aborted) of all transactions and is used to help atomically commit transactions and handle aborted transactions. The same copy of the data is used for both runtime access and durability. Multiple worker threads run on top of the storage layer to execute transactions, in which threads can read or update records in the tables under snapshot isolation.

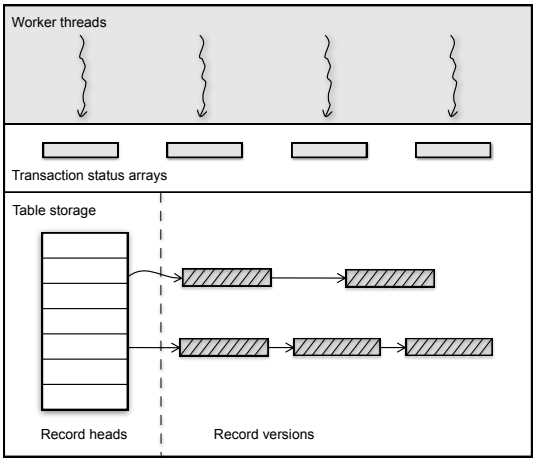


Fig. 1 Single-layer NVM storage of MCSI. Tables maintain multi-version records in NVM. Each worker thread has a persistent array that stores the commit status of their transactions, so we can identify uncommitted changes in the database.

3.2 Multi-version storage

MCSI uses multi-version storage. When a transaction needs to modify a record, it creates an updated new version instead of overwriting the existing version. Different versions of the same record are organized as a lock-free singly linked list in an append-only and newest-to-oldest fashion[26]. This singly linked list is called the version chain (Fig. 2).

Records in the table are indexed by implicit primary keys (record IDs). We locate a record by using its record head. The record head contains the latest_version pointer pointing to the latest record version. Each record version contains the payload data and a prev_version pointer pointing to its previous version in the chain. All the information is durably stored in NVM, i.e., it can survive after system crashes and restarts.

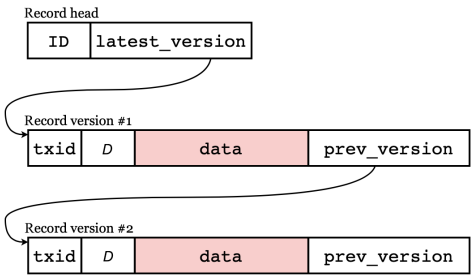


Fig. 2 Version chain of a record. Different versions of the record are organized as a newest-to-oldest linked list. Once created, the version data will be read-only. The txid is the unique identifier of the version's creator transaction, so we can look up the transaction status array to know if the version has been committed. The *D* field marks the version as a tombstone to indicate the record is deleted.

When a new version of a record is created, it is inserted into the front of the version chain, so the latest version pointer in the record head points to it instead. The content of inserted record versions will not be changed in the future. A tombstone version is used to indicate the removal of a record. Therefore, updating and removing operations are the same.

3.3 Identifying committed versions

A record version is committed if its creator transaction is committed. We need to identify committed versions in the storage layer for two reasons. One is that to ensure crash consistency, we have to identify and filter out versions inserted by uncompleted transactions to make sure we will not read them after recovery from a crash. The second reason is to implement snapshot isolation, in which a necessary requirement is that transactions can only see committed versions in the database.

To determine if a record version is committed, the status of its creator transaction is needed. We use a similar approach as PostgreSQL[17] and N2DB[14] with a simpler metadata structure. In the version chain, each record version contains its creator transaction's unique identifier txid. We use transaction status arrays (see Section 4.4) in NVM to store the status (initial, running, committed, and aborted) of each transaction.

The status can be atomically updated, which is why all its associated records can also change their states atomically. Other transactions can use such information to determine if versions are committed without a race condition. Also, being able to atomically commit all versions simplifies crash recovery because no write-ahead logging is needed.

The txid fields and status arrays are stored in NVM so we can correctly identify non-committed record versions after recovery from a crash. In MCSI, txids are two-dimensional and divided across different worker threads (see Section 4.2), so each worker thread has its own status array for its txid subset. This feature is different from the writer-shared commit log as used in PostgreSQL and N2DB.

4 Multi-Clock Concurrency Control

4.1 Snapshot isolation

MCSI implements snapshot isolation, an isolation level that is widely used in multi-version databases because it allows a high level of concurrency. Figure 3 depicts the relationship between snapshots and transactions in

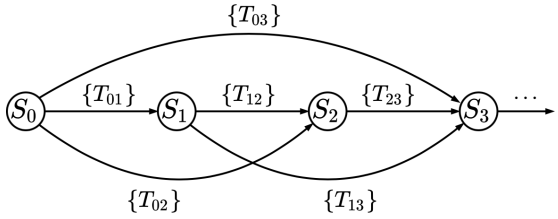


Fig. 3 Relationship between the snapshots and transactions in MCSI. Snapshots form a serial history of the database. The x-th snapshot is denoted as S_x. A transaction has to start from a snapshot S_i, and its changes may be reflected in a future snapshot S_j. All such transactions are denoted as $\{T_{ij}\}$.

MCSI. The snapshot (S_0, S_1, S_2, ...) describes the contents of the database at a specific time. S_0 is the initial empty state of the database. Each transaction sees a snapshot S_i that is generated before its start. If it is not a read-only transaction, then it will insert or modify one or several records by inserting newer versions to the version chains. Those modifications will be reflected in a future snapshot S_j after the transaction commits. We use $\{T_{ij}\}$ to denote all such transactions as indicated by the arrows in Fig. 3, and $C_{i,j}$ to denote their committed changes. Note that write-write conflicts on the same record are not allowed, i.e., a transaction T_{ij} operating on snapshot S_i is not allowed to commit if a record being updated by T_{ij} has been changed by another transaction since S_i was taken.

In this model, transactions commit their changes to the database between different snapshots, so all snapshots together form a serial history of the database. The snapshot S_x contains the contents of the previous snapshot S_{x-1} and the newly committed changes between S_{x-1} and S_x,

$$S_x = S_{x-1} + C_{0,x} + C_{1,x} + \cdots + C_{x-1,x} \quad (1)$$

from which we can induce that

$$S_x = S_0 + \sum_{0<j\leqslant x} \sum_{0\leqslant i<j} C_{i,j} \quad (2)$$

i.e., snapshot S_x consists of inserted changes from all committed transactions before the snapshot time x. Furthermore, the committed changes $C_{i,j}$ in the multi-version storage can be represented by the changes from currently completed (including committed and aborted) transactions $F_{i,j}$ excluding the aborted changes $A_{i,j}$,

$$S_x = S_0 + \sum_{0<j\leqslant x} \sum_{0\leqslant i<j} F_{i,j} - \sum_{0<j\leqslant x} \sum_{0\leqslant i<j} A_{i,j} \quad (3)$$

As a result, in the multi-version storage, we do not need to create actual physical copies of the database to represent snapshots. Instead, we can record all completed (including committed and aborted) transactions in the snapshot and filter out aborted ones by using the method described in Section 3.3 (this is exactly what PostgreSQL achieved). On the single-layer NVM storage engine, this kind of snapshot implementation has to consider in the following:

• **Concurrent transactions start during snapshot generation.** During the generation of a snapshot, newly started transactions should be carefully handled so they do not end up recorded as completed transactions in the generated snapshot.

• **All snapshots together should form a serial history of the database.** All generated snapshots in the system should fulfill the requirement in Eq. (1). Transactions from different threads may have their own snapshots at the same time, and we need to ensure that, for any two snapshots, one of them strictly contains the other.

• **Transaction status storage,** i.e., how to implement the data structure that stores the status of each transaction in NVM so that we can filter out aborted versions when reading the multi-version storage.

Table 2 describes the main differences of the snapshot isolation between PostgreSQL and MCSI. The PostgreSQL approach is inefficient for the single-layer NVM database storage engine. MCSI tries to eliminate the snapshot generation bottleneck by using per-thread clocks as discussed below.

4.2 Multi-clock timestamp allocation

In MCSI, each worker thread has its own local clock which is a 54-bit monotonic increasing counter. When a transaction starts, it is assigned a unique identifier

Table 2 Overview of different design choices between the PostgreSQL-style snapshot isolation and MCSI.

	Concurrent transactions start during snapshot	Ensuring serial history of snapshots	Transaction status storage
PostgreSQL	Snapshot generation scans a global list to exclude active transactions from the list, and at the same time prevents new transactions from starting.	Snapshots are generated one after another strictly.	A write-shared commit log to store transaction status for all worker threads.
MCSI	By using multi-clock timestamp assignment, snapshot generation scans the last completed timestamps of each thread, which allows new concurrent transactions to start.	Generated snapshots are buffered in the system. Concurrent attempts to require a snapshot can use a buffered one.	Each worker thread has a single-writer status array to store the status of transactions executed in this thread.

called txid, a 64-bit unsigned integer that is made up of two parts. The first 10 bits are the clock_id, indicating the thread in which the transaction is executed. The remaining 54 bits, clock_ts, are the clock value of the worker thread when the transaction is started. After a txid is generated, the corresponding thread's local counter will be incremented by one. We use T(clock_id, clock_ts) to represent a transaction. When a transaction is running, it uses the assigned txid to mark its inserted, updated, and removed tuple versions.

Essentially, the txids from all threads form vector clocks. We cannot determine the dependency between transactions by comparing their txids. Assigning multi-clock timestamps in this way helps us efficiently record currently completed transactions without blocking new transactions from starting when generating snapshots.

4.3 Snapshot generation

MCSI records the information of currently completed transactions from all threads to describe a snapshot. Each thread assigns monotonic increasing clock_ts to its transactions and executes them one after another; thus, the range of all currently completed transactions on each worker thread can be described by the local timestamp (clock_ts) of the last completed transaction in that thread (last completed timestamp).

We maintain a global completed timestamp array to store the current last completed timestamps for each thread. The timestamps are simply stored as 64-bit unsigned integers, but each of them is aligned to the cache line size of the CPU to eliminate false sharing. When a thread completes a transaction, it updates its last completed timestamp in the array with an atomic write instruction. When MCSI creates a snapshot, it scans all the timestamps in the array by using atomic read instructions. The created snapshots S can be essentially seen as vector clocks: $S = [ts_0, ts_1, ts_2, \ldots]$, as depicted in Fig. 4.

Scanning the entire completed timestamp array for generating a new snapshot is not atomic. When another thread is updating its last completed timestamp, two concurrently generated snapshots in this situation may not form a serial history. To deal with this issue, we forbid concurrent snapshot generation, which is why concurrent update and scan operations on the global completed timestamp array are serialized. This approach is sufficient to ensure that the relationships between transactions and generated snapshots satisfy Eq. ((1)).

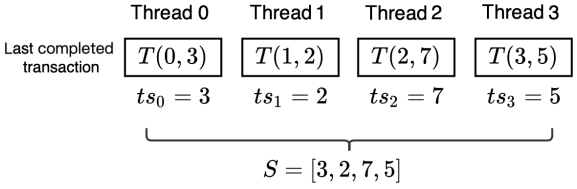


Fig. 4 Snapshot vector clock. MCSI records the local timestamp of every worker thread's last completed transaction in a snapshot.

Figure 5a demonstrates an example with two worker threads. We use $[ts_0, ts_1]$ to denote their last completed timestamps. Thread 0 completed transaction $T(0,4)$ and is updating its last completed timestamp ts_0 from 3 to 4. Thread 1 completed transaction $T(1,3)$ and is updating ts_1 from 2 to 3. The two transactions are both based on the previous snapshot $S_2 = [3,2]$. At the same time, snapshot S_3 is being generated concurrently. The update from worker thread 1 is captured but the update from worker thread 0 is not, so the generated $S_3 = [3,3]$. The updated ts_0 is delayed to a later snapshot $S_4 = [3,4]$. The relationship between S_2, S_3, S_4, and the two transactions fulfill the requirements of snapshot isolation described in Section 4.1, and is shown in Fig. 5b.

For comparison, PostgreSQL's snapshot generation blocks concurrent transactions from starting. PostgreSQL uses a single global clock to generate transaction timestamps and maintain a global active transaction

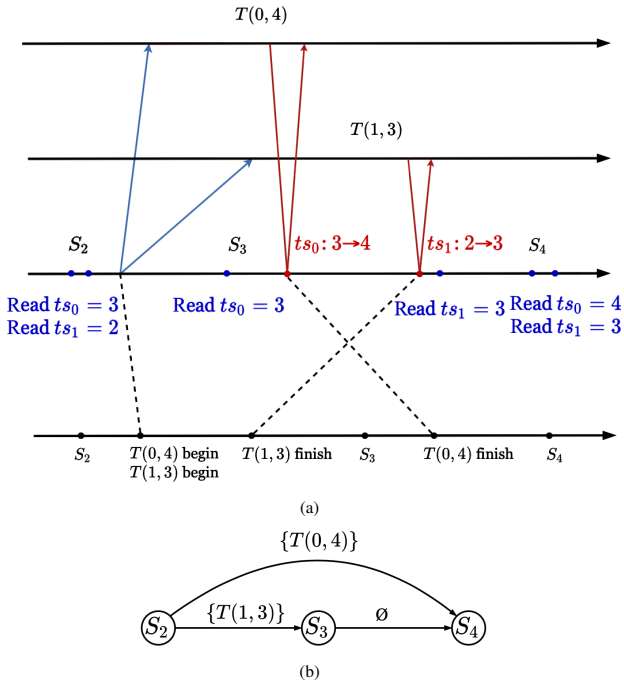


Fig. 5 Example of snapshot generation with concurrent transaction commits. S_3's scanning of the completed timestamp array captures $T(1,3)$'s commit, but $T(0,4)$'s commit is not captured and delayed to a later snapshot S_4. This behavior does not violate the model of snapshot isolation described in Section 4.1.

list at runtime. The current completed transactions in its snapshot are all transactions that have a timestamp less than the current global clock value, excluding the running ones in the global active transaction list. When a snapshot is generated, a race condition exists in when a concurrent newly started transaction increases the global timestamp to allocate its timestamp but has not inserted this timestamp to the active transaction list, and the generated snapshot may incorrectly record it as a completed transaction. To prevent this situation, PostgreSQL has to block new transactions from starting during snapshot generation.

Another design choice in MCSI is snapshot buffering. When a transaction starts, it needs to obtain a snapshot by scanning the global completed timestamp array. However, as said before, concurrent snapshot generation is not allowed, which is why we have to consider the situation where a concurrent transaction also wants to obtain a snapshot at the same time. With snapshot buffering, the process of actual snapshot generation is protected by a mutex that prevents concurrent scanning of the global completed timestamp array. The database maintains a global buffer to store generated snapshots for concurrent transactions to use. Consider an example with three transactions (T_A, T_B, and T_C). All of them want to acquire a snapshot at their start time, and they perform in the following timeline:

• For transaction T_A, the try-lock succeeds. It generates a snapshot, appends it to the global snapshot buffer, and unlocks the mutex.

• Transaction T_B starts after transaction T_A is finished. The try-lock also succeeds. It then proceeds to generate a new snapshot, which will reflect the latest state of the database.

• Transaction T_C starts concurrently with transaction T_B. Transaction T_B is already in the process of generating a snapshot; thus, the try-lock will fail. In this situation, transaction T_C will use the previously saved snapshot (generated by transaction T_A) in the global buffer instead.

• Transaction T_B finishes generating the new snapshot, appends it to the global snapshot buffer, and unlocks the mutex.

With snapshot buffering, transaction T_C can use the buffered snapshot generated by an earlier transaction, so it does not have to wait for the snapshot generation of transaction T_B to finish.

4.4 Transaction status arrays

To identify uncommitted record versions, we need to check the status of their creator transactions. MCSI stores this information by using status arrays in NVM. As Fig. 6 depicts, each worker thread has its own status array. The 2-bit transaction status (initial, running, committed, and aborted) is packed into 64-bit words (actually for 31 transactions, as explained below), and we use a read-modify-write instruction to update each of them. A status array will be modified only by its owner thread but may be read by other threads, i.e., Multiple Readers, Single Writer (MRSW).

To ensure consistency, write and read operations on the status array should be durably linearizable[30], i.e., linearizable even when a crash occurs. Therefore, the status array should make sure that the readers always read the persisted status. However, when writing to the status array, we need to use additional cache line flush instructions, such as CLWB, to ensure data are persistent in NVM because a store instruction buffers the data in the CPU cache. These two steps are not atomic; thus, a reader can read dirty information in between. To solve this problem, we adopt the helping mechanism[31]. In the status array, each 64-bit word has one dirty bit to indicate whether it needs to be persisted. If a reader finds that it has read a word with the dirty bit, then it issues a cache flush instruction to help persist the word before the result is returned to guarantee that persisted data are always read. Algorithm 1 shows the pseudocode of setting and getting transaction status on a persistent 64-bit word. The most significant bit of each 64-bit word in the status array is the dirty bit (Lines 1 and 2). The remaining bits can hold the 2-bit status for up to 31 transactions.

For a status set operation, the writer provides an offset (which can be calculated from the transaction txid) that indicates the position of the bits to manipulate and a 2-bit status value to write (Line 3). The writer reads the previous value of the word (Line 5), changes the

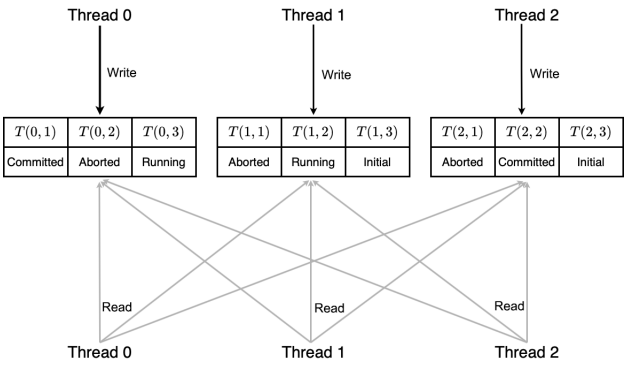


Fig. 6 MRSW status arrays in MCSI. The status of each transaction takes up 2 bits.

Algorithm 1 Persistent 64-bit MRSW word

```
1  uint64_t word;
2  const DIRTY_BIT = 1 << 63;
3  function set_status(offset, status)
4      mask = 0b11 << offset;
5      old_val = word.load();
6      new_val = (old_val & ~mask) | (status << offset);
7      word.store(new_val | DIRTY_BIT);
8      cache_flush(&word);
9      word.store(new_val & ~DIRTY_BIT);
10 function get_status(offset)
11     mask = 0b11 << offset;
12     res = word.load();
13     status = (res & mask) >> offset;
14     if (res & DIRTY_BIT) {
15         cache_flush(&word);
16     }
17     return status;
```

corresponding status bits without affecting other bits (Line 6), and writes the new value back with the dirty bit set to 1 (Line 7). This process indicates that this word has been changed but not persisted. The writer then persists the word by using a cache flush instruction (Line 8) and clears the dirty bit (Line 9).

For a status get operation, the reader also provides the offset for the status bits in the word (Line 10). The reader reads the value of the word (Line 12) and extracts the corresponding status bits (Line 13). Before it returns, the reader needs to check the dirty bit of the word it just read (Line 14). If the dirty bit is 1, then the reader may have read a yet-not-persisted value wrote by a concurrent status set operation. In this case, the reader helps persist the word it just read (Line 15) before the status is returned.

With only one writer thread on each status array, no concurrent updates can occur on the same 64-bit word. Therefore, the read and write operations on our status arrays are wait-free. They are much faster than the commit log in PostgreSQL that is write-shared by all threads.

For each transaction, once it reached the committed or aborted status, its status is finalized and will not change anymore. We then inline these two final statuses in the record versions to reduce unnecessary status array reading.

4.5 Transaction execution

This section describes the transaction execution protocols in MCSI.

Transaction start. When a worker thread wants to start a transaction, it first generates a txid for the transaction and sets the transaction status to running in the status array. Then, it acquires a snapshot of the database either by scanning the last completed timestamp of each worker thread or grabbing the most recently generated one in the global snapshot buffer.

Reading a record. With snapshot isolation, when a transaction wants to read a record, it needs to find the correct version in the version chain of this record by using the visibility rule described below. It first determines which record versions are inserted before its snapshot was taken by comparing their txids with the snapshot vector clock. The transaction chooses the first committed version in them as the version to read. An example is shown in Fig. 7. If no visible version is found or the visible version is a tombstone that indicates that the record has been deleted, then the record does not exist for this transaction.

Updating a record. With snapshot isolation, we need to prevent conflicting updates on the same record. For a transaction to successfully update a record, no other committed version must exist after its visible version in the version chain. We also assume a version inserted by another running transaction will soon commit. Under such conditions, the updating operation will conflict and the later transaction has to abort. Examples are given in Fig. 8.

This checking process and the actual update operation are also not atomic. To deal with this situation, an updating transaction first reads the latest version pointer in the record head before the check. If it passes the check, then the transaction creates an updated record version and atomically inserts it to the front of the version chain. The transaction first properly sets the previous version pointer of the new record version, then atomically changes the latest version pointer of the record head to the new version by using a Compare-And-Swap (CAS) instruction, as depicted in Fig. 9. The CAS instruction compares against the latest version pointer the transaction read before the check and if it fails, then it means a concurrent update has been inserted after the

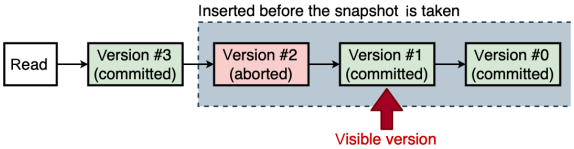


Fig. 7 Finding the visible version when reading a record. A transaction will read the first committed version inserted before its snapshot is taken.

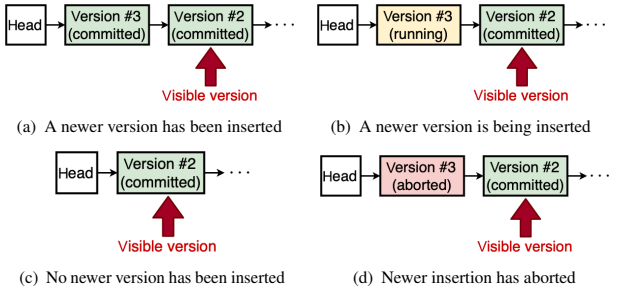


Fig. 8 Preventing conflicting updates on the same record. If a transaction wants to update the record, then we ensure no newer committed version is present in the version chain. We also assume a version inserted by a running transaction will soon commit. The updating transaction will abort in cases (a) and (b), and continue in cases (c) and (d).

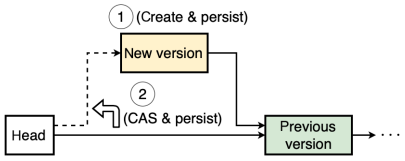


Fig. 9 Atomically insert an updated record version to the version chain. We create a new version that contains the updated data and points to the previous version in the chain. The new version is persisted in NVM and we insert it to the front of the version chain by using a CAS instruction. Finally, we persist the latest version pointer of the record head.

check and the current transaction has to abort.

To ensure durability and crash consistency, we persist the data and pointers after they are changed. Before Step ② in Fig. 9, we persist the whole new version (including the previous version pointer) to NVM by using the CLWB instruction. After Step ②, we persist the latest version pointer of the record head. If a crash happens, then the latest version pointer in NVM either points to the previous version or the new version; in either case, the entire version chain is crash consistent.

Committing and aborting. For read-only transactions that make no changes in the database, their committing and aborting are just no-ops. For a transaction that has changes in the database, we have to set its status to committed or aborted in the status array, then publish its clock_ts as the last completed timestamp of its worker thread. The vector clock in later generated snapshots will include this transaction.

5 Evaluation

In this section, we evaluate the performance of MCSI. To compare it with a traditional concurrency control scheme from two-layer databases, we use the NVM database N2DB[14], which adapts the concurrency control implementation from PostgreSQL to single-layer NVM storage. Experimental evaluation shows the benefits of MCSI; when the worker thread number is low, it has a similar performance as N2DB, but MCSI is much more scalable when the thread number increases.

All experiments use a single server equipped with Intel Xeon Gold 5220 CPUs. Each socket has 6 Intel Optane Persistent Memory modules attached with an interleaved configuration. The remote access of NVM across NUMA nodes is extremely slow[21], which is why we run the evaluation on a single socket only.

5.1 YCSB experiments

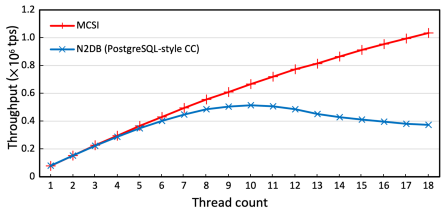
We run the YCSB[32] benchmark to highlight the scalability of MCSI's snapshot isolation implementation. YCSB is a widely used benchmark in key-value storage systems and is also used to evaluate transactional databases. The YCSB benchmark uses a single table of 1 million tuples, each with a primary key and 10 string columns. The size of each string column is 100 bytes, so the size of each tuple is about 1 KB. The total size of the table is about 1 GB. In our experiment, each transaction randomly chooses 1 to 5 records in the table and performs read or update operations. The original YCSB test defines 5 workloads; the first three workloads *A*, *B*, and *C* are used to evaluate MCSI under different read/write ratios (Table 3). Workloads *D* and *E* are not implemented because functions such as range scans are not yet implemented in our storage engine.

We measured the throughput (transaction per second) under different thread counts and the results are shown in Fig. 10. When only one worker thread is present, MCSI and N2DB achieve about the same performance in all three workloads. However, the performance of MCSI scales well as the number of threads increases, while N2DB achieves its maximum throughput at 7 to 10 threads, after which the performance drops. The maximum transaction throughputs that can be achieved by MCSI and N2DB are shown in Table 4. The maximum throughput of MCSI is improved by 101% to 195% compared with N2DB in our YCSB test.

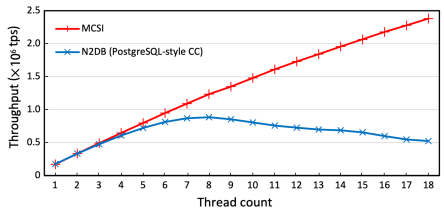
Two factors make MCSI faster than the PostgreSQL-style concurrency control scheme in N2DB: snapshot

Table 3 Different YCSB workloads.

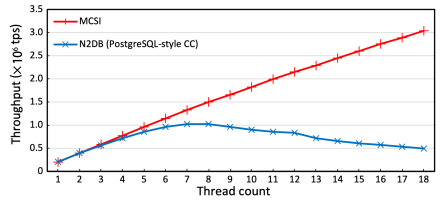
Workload	Transaction operation
A – Update heavy	50% reads, 50% updates
B – Read heavy	95% reads, 5% updates
C – Read only	100% reads



(a) Workload *A*



(b) Workload *B*



(c) Workload *C*

Fig. 10 YCSB throughput results.

Table 4 Maximum throughput (YCSB).

Workload	MCSI ($\times 10^6$ tps)	N2DB ($\times 10^6$ tps)	Improvement (%)
A	1.035 (18 threads)	0.515 (10 threads)	101
B	2.379 (18 threads)	0.886 (8 threads)	169
C	3.026 (18 threads)	1.025 (7 threads)	195

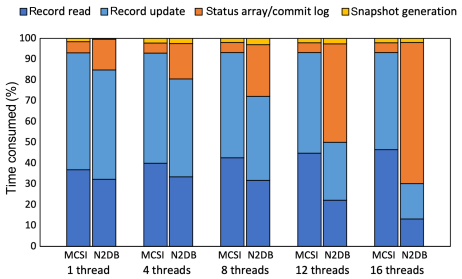
generation and the data structure for transaction status storage in NVM. With multi-clock and snapshot buffering, MCSI's snapshot generation is asynchronous and does not block concurrent transaction starts. However, the PostgreSQL-style snapshot generation process will block concurrent transaction starts. Therefore, it is not scalable.

For the transaction status storage, MCSI uses per-thread transaction status arrays in NVM. Each of them has only a single writer, so no write contention occurs on them as the thread number increases. N2DB adopted the writer-shared commit log from PostgreSQL. The PostgreSQL-style concurrency control scheme uses a central global clock to generate transaction timestamps. As a result, concurrent transactions are more likely to have timestamps that are close to each other, hence being more likely to write to adjacent locations in the commit log when they are committing or aborting. This behavior leads to write contention and false-sharing problems in the CPU cache.

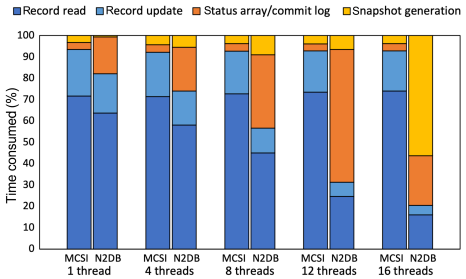
To derive some insight into the performance problems of the PostgreSQL-style concurrency control scheme, we profiled MCSI and N2DB under different worker thread numbers. The detailed performance breakdown is shown in Fig. 11. As we can see, the commit log in NVM accounts for the largest overhead in N2DB as the thread number increases. However, Workloads *B* and *C* have many read-only transactions that do not make changes to the database. Therefore, they do not need to update their status in the commit log when they complete, putting less pressure on the commit log than Workload *A*. This feature highlights the performance issues with its snapshot generation at 16 threads.

5.2 TPC-C experiments

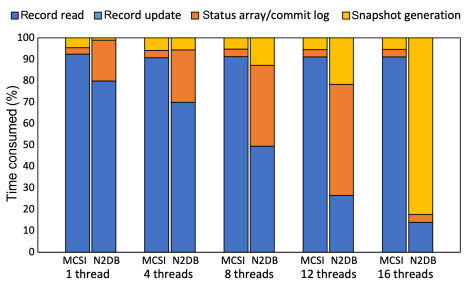
TPC-C[33] is an OnLine Transaction Processing (OLTP) benchmark for database systems. Transactions in TPC-C are much more complicated than those in YCSB. Each of our TPC-C experiments consists of 45% New-Order and 55% Payment transactions. A configurable number of warehouses is available in the TPC-C benchmark. Each worker thread mostly interacts with its local warehouse,



(a) Workload *A*



(b) Workload *B*

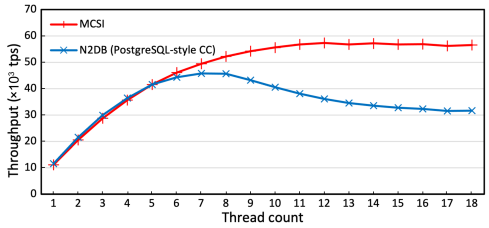


(c) Workload *C*

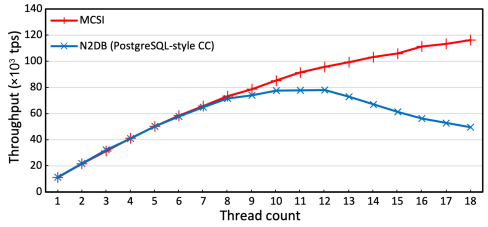
Fig. 11 YCSB performance breakdown.

but 10% of New-Order and 15% of Payment transactions access a remote warehouse. By adjusting the number of warehouses, we can control the workload contention. We evaluate three different contention configurations: 1 warehouse, 4 warehouses, and the uncontended configuration where the number of warehouses equals the worker thread number. The results are shown in Fig. 12.

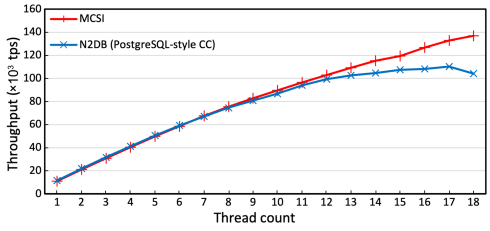
With snapshot isolation, transactions are more likely to abort under higher contentions because of write conflicts. For the contended workloads, snapshot generation and commit log writing of aborted transactions in N2DB has negative performance impacts on concurrent transactions. As a result, the total throughput drops when the thread number increases. In MCSI, the snapshot buffering and single-writer status maps can tolerate the extra overhead of aborted transactions. Thus, the throughput remains relatively stable as more worker threads are added. Also, for the uncontended configuration, MCSI can scale well, while N2DB encounters the same problems in the YCSB tests; thus, its throughput eventually decreases. The maximum transaction throughput that MCSI and N2DB can achieve is shown in Table 5. The maximum throughput of MCSI is 25% to 49% higher than that of N2DB in the TPC-C test.



(a) 1 warehouse



(b) 4 wareshouses



(c) uncontended (warehouses = thread count)

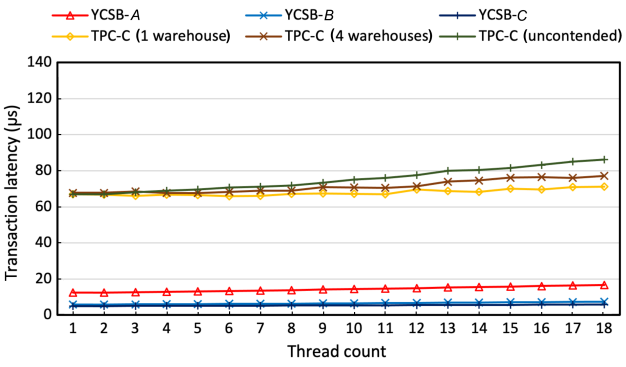
Fig. 12 TPC-C throughput.

Table 5 Maximum throughput (TPC-C).

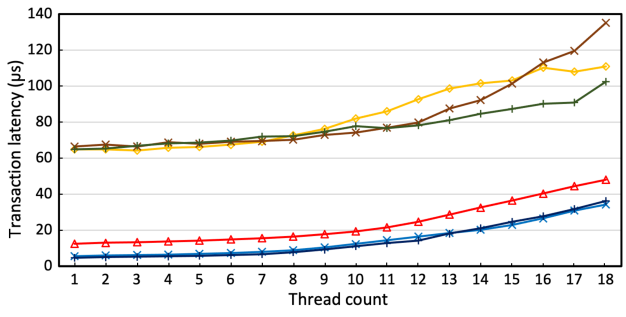
Workload	MCSI ($\times 10^3$ tps)	N2DB ($\times 10^3$ tps)	Improvement (%)
1 warehouse	57.38 (12 threads)	45.82 (7 threads)	25
4 warehouses	116.15 (18 threads)	78.08 (12 threads)	49
uncontended	137.57 (18 threads)	110.28 (17 threads)	25

5.3 Latency experiments

Having low transaction latency is important for an NVM database because NVM has a lower write latency than disks. We tested the average latency of transactions in the YCSB and TPC-C workloads with different worker thread counts. MCSI and N2DB have about the same low transaction processing latency in all workloads when only one thread is present. The measured latency includes the time of persisting modified data and transaction status to NVM. As Fig. 13 depicts, when the thread number increases, the transaction processing latency in MCSI remains relatively low, but the latency in N2DB increases by several times. Table 6 presents a comparison of their average transaction latency with 18 worker threads. The average latency of MCSI is 65% to 84% lower than that of N2DB in the YCSB workloads and 16% to 43% lower in the TPC-C workloads. The latency results show the advantage of MCSI's low-overhead snapshot generation and transaction status arrays on NVM.



(a) MCSI latency



(b) N2DB latency

Fig. 13 Transaction latency results.

Table 6 Transaction latency with 18 threads.

Workload	MCSI (μs)	N2DB (μs)	Reduced (%)
YCSB-*A*	16.7	48.1	65
YCSB-*B*	7.4	34.4	78
YCSB-*C*	5.8	36.2	84
TPC-C (1 warehouse)	71.3	111.0	36
TPC-C (4 warehouses)	77.1	135.2	43
TPC-C (uncontended)	86.2	102.5	16

6 Related Work

Serializable certifiers: Our protocol implements snapshot isolation. Existing research has discussed serializable certifiers, which can be used to implement the serializable isolation level on top of weaker isolation levels. SSI[19] is an algorithm that ensures serializability on snapshot isolation by checking "dangerous structures" in a dependency graph and aborting involved transactions. Neumann et al.[24] used a technique called precision locking to guarantee serializability on top of snapshot isolation in HyPer. Wang et al.[20] proposed serial safety net, which is a serializable certifier that can be applied on top of various isolation levels (such as snapshot isolation and read committed). Our work can also be extended to support serializability by using serializable certifiers.

Databases designed for NVM: In recent years, researchers have proposed new NVM-based database designs. Zen[29] is a log-free OLTP engine for NVM. Zen has a two-layer storage design. It maintains a multi-version tuple heap in NVM and uses tuple-level caching in DRAM. Unlike N2DB and MCSI, the two-layer storage design of Zen decouples concurrency control from durability and crash consistency. Zen runs the concurrency control in the DRAM rather than directly on NVM and ensures the atomicity of committed transactions by appending modified tuples to the NVM tuple heap and atomically setting a Last Persisted (LP) bit for the transaction. The LP bits serve a similar purpose of atomically committing transactions as the commit log in N2DB and the status arrays in MCSI. MCSI performs concurrency control directly on NVM; thus, the status arrays are also used to determine uncommitted data at run time.

Snapshot isolation concurrency control in NVM: Snapshot isolation has also been used in other NVM-related works. Pisces[34] is a persistent transactional memory that leverages snapshot isolation in NVM to improve concurrency. It uses Dual-Version Concurrency Control (DVCC), i.e., only two versions of each object are kept to limit the searching time in the version chain. However, if long-running transactions take place, then write operations in Pisces will be delayed because the snapshot held by long-running transactions will prevent the DVCC versions from being recycled. Pisces also generates timestamps with a single clock and uses the commit timestamp on versions to determine if they are in the snapshot. Therefore, when a transaction commits, it has to atomically update the timestamp of the versions it created, during which concurrent read operations are blocked.

7 Conclusion and Future Work

This paper presents MCSI, a snapshot isolation implementation that can eliminate the performance bottlenecks of the PostgreSQL-style concurrency control scheme in the single-layer NVM database. MCSI uses multi-clock timestamp allocation, so snapshots can be represented as vector clocks. MCSI scans a global completed timestamp array to generate snapshots. With the use of snapshot buffering, transaction starts will not be blocked by a concurrent snapshot generation. Multi-clock also enables per-thread transaction status arrays; each of them has only a single writer and is more efficient than a write-shared commit log. The maximum transaction throughput of MCSI is 101%–195% higher than that of N2DB for the YCSB workloads, and 25%–49% higher for the TPC-C workloads. Moreover, the transaction latency of MCSI remains relatively stable as the thread count increases. With 18 worker threads, the average transaction latency of MCSI is 65%–84% lower than that of N2DB for the YCSB workloads, and 16%–43% lower for the TPC-C workloads.

As NVM technologies develop, how to use them to build new databases will be an interesting topic for the academia and the industry. Our future work includes implementing serializable certifiers in MCSI to achieve a stronger isolation level. We will also study the NUMA effects in MCSI and use MCSI as a building block for distributed NVM databases.

Acknowledgment

This work was supported by the National Key Research & Development Program of China (No. 2016YFB1000504) and the National Natural Science Foundation of China (Nos. 61877035, 61433008, 61373145, and 61572280).

References

[1] M. Andrei, C. Lemke, G. Radestock, R. Schulze, C. Thiel, R. Blanco, A. Meghlan, M. Sharique, S. Seifert, S. Vishnoi, et al., SAP HANA adoption of non-volatile memory, *Proceedings of the VLDB Endowment*, vol. 10, no. 12, pp. 1754–1765, 2017.

[2] A. Eisenman, D. Gardner, I. AbdelRahman, J. Axboe, S. Y. Dong, K. Hazelwood, C. Petersen, A. Cidon, and S. Katti, Reducing DRAM footprint with NVM in Facebook, in *Proc. 13th EuroSys Conf.*, Porto, Portugal, 2018, p. 42.

[3] J. H. Kim, J. Kim, H. Kang, C. G. Lee, S. Park, and Y. Kim, pNOVA: Optimizing shared file I/O operations of NVM file system on manycore servers, in *Proc. 10th ACM SIGOPS Asia-Pacific Workshop on Systems*, Hangzhou, China, 2019, pp. 1–7.

[4] J. DeBrabant, J. Arulraj, A. Pavlo, M. Stonebraker, S. Zdonik, and S. R. Dulloor, A prolegomenon on OLTP database systems for non-volatile memory, in *ADMS'14*, Hangzhou, China, 2014, pp. 57–63.

[5] A. van Renen, V. Leis, A. Kemper, T. Neumann, T. Hashida, K. Oe, Y. Doi, L. Harada, and M. Sato, Managing non-volatile memory in database systems, in *Proc. 2018 Int. Conf. Management of Data*, Houston, TX, USA, 2018, pp. 1541–1555.

[6] R. Fang, H. I. Hsiao, B. He, C. Mohan, and Y. Wang, High performance database logging using storage class memory, in *2011 IEEE 27th Int. Conf. Data Engineering*, Hannover, Germany, 2011, pp. 1221–1231.

[7] S. Gao, J. L. Xu, B. S. He, B. Choi, and H. B. Hu, PCMLogging: Reducing transaction logging overhead with PCM, in *Proc. 20th ACM Int. Conf. Information and Knowledge Management*, Glasgow, UK, 2011, pp. 2401–2404.

[8] T. Z. Wang and R. Johnson, Scalable logging through emerging non-volatile memory, *Proceedings of the VLDB Endowment*, vol. 7, no. 10, pp. 865–876, 2014.

[9] J. Arulraj, J. Levandoski, U. F. Minhas, and P. A. Larson, BzTree: A high-performance latch-free range index for non-volatile memory, *Proceedings of the VLDB Endowment*, vol. 11, no. 5, pp. 553–565, 2018.

[10] X. J. Zhou, L. D. Shou, K. Chen, W. Hu, and G. Chen, DPTree: Differential indexing for persistent memory, *Proceedings of the VLDB Endowment*, vol. 13, no. 4, pp. 421–434, 2019.

[11] S. N. Ma, K. Chen, S. M. Chen, M. X. Liu, J. L. Zhu, H. B. Kang, and Y. W. Wu, ROART: Range-query optimized persistent ART, in *19th USENIX Conf. File and Storage Technologies*, Santa Clara, CA, USA, 2021, pp. 1–16.

[12] I. Oukid, D. Booss, W. Lehner, P. Bumbulis, and T. Willhalm, SOFORT: A hybrid SCM-DRAM storage engine for fast data recovery, in *Proc. 10th Int. Workshop on Data Management on New Hardware*, Snowbird, UT, USA, 2014, p. 8.

[13] J. Arulraj, M. Perron, and A. Pavlo, Write-behind logging, *Proceedings of the VLDB Endowment*, vol. 10, no. 4, pp. 337–348, 2016.

[14] M. Liu, Concurrency control for non-volatile memory systems, (in Chinese), PhD dissertation, Department of Computer Science and Technology, Tsinghua University, Beijing, China, 2020.

[15] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil, A critique of ANSI SQL isolation levels, *ACM SIGMOD Record*, vol. 24, no. 2, pp. 1–10, 1995.

[16] Oracle Database, https://www.oracle.com/database/, 2020.

[17] PostgreSQL, https://www.postgresql.org/, 2020.

[18] Microsoft SQL Server, https://www.microsoft.com/sql-server, 2020.

[19] M. J. Cahill, U. Röhm, and A. D. Fekete, Serializable isolation for snapshot databases, *ACM Transactions on Database Systems*, vol. 34, no. 4, p. 20, 2009.

[20] T. Z. Wang, R. Johnson, A. Fekete, and I. Pandis, Efficiently making (almost) any concurrency control mechanism serializable, *The VLDB Journal*, vol. 26, no. 4, pp. 537–562, 2017.

[21] J. Yang, J. Kim, M. Hoseinzadeh, J. Izraelevitz, and S. Swanson, An empirical guide to the behavior and use of scalable persistent memory, in *18th USENIX Conf. File and Storage Technologies*, Santa Clara, CA, USA, 2020, pp. 169–182.

[22] M. Grund, J. Krüger, H. Plattner, A. Zeier, P. Cudre-Mauroux, and S. Madden, Hyrise: A main memory hybrid storage engine, *Proceedings of the VLDB Endowment*, vol. 4, no. 2, pp. 105–116, 2010.

[23] C. Diaconu, C. Freedman, E. Ismert, P. A. Larson, P. Mittal, R. Stonecipher, N. Verma, and M. Zwilling, Hekaton: SQL server's memory-optimized OLTP engine, in *Proc. 2013 ACM SIGMOD Int. Conf. Management of Data*, New York, NY, USA, 2013, pp. 1243–1254.

[24] T. Neumann, T. Mühlbauer, and A. Kemper, Fast serializable multi-version concurrency control for main-memory database systems, in *Proc. 2015 ACM SIGMOD Int. Conf. Management of Data*, Melbourne, Australia, 2015, pp. 677–689.

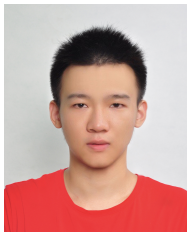
[25] J. Lee, M. Muehle, N. May, F. Faerber, V. Sikka, H. Plattner, J. Krueger, and M. Grund, High-performance transaction processing in SAP HANA, *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, vol. 36, no. 2, pp. 28–33, 2013.

[26] Y. J. Wu, J. Arulraj, J. X. Lin, R. Xian, and A. Pavlo, An empirical evaluation of in-memory multi-version concurrency control, *Proceedings of the VLDB Endowment*, vol. 10, no. 7, pp. 781–792, 2017.

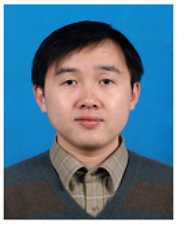
[27] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz, ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging, *ACM Transactions on Database Systems*, vol. 17, no. 1, pp. 94–162, 1992.

[28] H. Kimura, FOEDUS: OLTP engine for a thousand cores and NVRAM, in *Proc. 2015 ACM SIGMOD Int. Conf. Management of Data*, Melbourne, Australia, 2015, pp. 691–706.

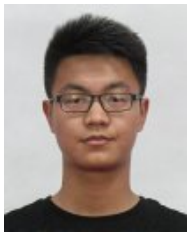
[29] G. Liu, L. Y. Chen, and S. M. Chen, Zen: A high-throughput log-free OLTP engine for non-volatile main memory, *Proceedings of the VLDB Endowment*, vol. 14, no. 5, pp. 835–848, 2021.
[30] J. Izraelevitz, H. Mendes, and M. L. Scott, Linearizability of persistent memory objects under a full-system-crash failure model, in *Int. Symp. Distributed Computing*, Paris, France, 2016, pp. 313–327.
[31] T. David, A. Dragojevi, R. Guerraoui, and I. Zablotchi, Log-free concurrent data structures, in *Proc. 2018 USENIX Annu. Technical Conf.*, Boston, MA, USA, 2018, pp. 373–385.
[32] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, Benchmarking cloud serving systems with YCSB, in *Proc. 1st ACM Symp. Cloud Computing*, Indianapolis, IN, USA, 2010, pp. 143–154.
[33] TPC benchmark C, http://www.tpc.org/tpcc/, 2010.
[34] J. Y. Gu, Q. Q. Yu, X. Y. Wang, Z. G. Wang, B. Y. Zang, H. B. Guan, and H. B. Chen, Pisces: A scalable and efficient persistent transactional memory, in *2019 USENIX Annu. Technical Conf.*, Renton, WA, USA, 2019, pp. 913–928.



Xuyang Liu received the BEng degree from Beihang University, Beijing, China in 2018. Currently, he is a master student at the Department of Computer Science and Technology, Tsinghua University, China. His research interests include non-volatile memory and transaction processing in database systems.



Kang Chen received the PhD degree in computer science and technology from Tsinghua University, China in 2004. Currently, he is an associate professor of computer science and technology at Tsinghua University. His research interests include parallel computing, distributed processing, and cloud computing.



Mengxing Liu received the BEng degree in computer science and technology from Tsinghua University, China in 2015. Currently, he is a PhD candidate at Tsinghua University. His research interests include transactional systems and concurrent data structures and how they are impacted by new architectural technology, such as non-volatile memory and hardware transactional memory.

Shiyu Cai received the BEng degree from Shanghai Jiao Tong University, China in 2018. Currently, he is a master student at the Department of Computer Science and Technology, Tsinghua University, China. His research interests include non-volatile memory, database systems, and distributed system.

Yongwei Wu received the PhD degree in applied mathematics from the Chinese Academy of Sciences in 2002. He is currently a professor in computer science and technology at Tsinghua University, China. His research interests include distributed processing, virtualization, and cloud computing. He has published over 80 research publications and has received two Best Paper Awards. He is currently on the editorial board of the *International Journal of Networked and Distributed Computing* and *Communication of China Computer Federation*. He is a member of the IEEE.

Weimin Zheng received the MEng degree in computer science from Tsinghua University, Beijing, China in 1982. Currently, he is a professor at the Department of Computer Science and Technology, Tsinghua University, Beijing, China. His research interests include high performance computing, network storage, and parallel compiler. He is an academician of Chinese Academy of Engineering.