Noname manuscript No.
(will be inserted by the editor)

RCBench: An RDMA-enabled Transaction Testbed For Analyzing Concurrency Control Algorithms

Technical Report

Hongyao Zhao · Jingyao Li · Wei Lu · Qian Zhang · Wanqing Yang · Jiajia Zhong · Meihui Zhang · Haixiang Li · Xiaoyong Du · Anqun Pan.

the date of receipt and acceptance should be inserted later

Abstract Distributed transaction processing over the TCP/IP network suffers from the *weak transaction scalability* problem, i.e., its performance drops significantly when the number of involved data nodes per transaction increases. Although quite a few of works over the high-performance RDMA-capable network are proposed, they mainly focus on accelerating distributed transaction processing, rather than solving the weak transaction scalability problem. In this paper, we propose *RCBench*, an RDMA-enabled transaction testbed, which serves as a unified evaluation tool for assessing the transaction scalability of various concurrency control algorithms. The usability and advancement of RCBench primarily come from the proposed concurrency control primitives based on RDMA, which facilitate the convenient implementation of RDMA-enabled concurrency control algorithms. Various optimization principles are proposed to ensure that concurrency control algorithms in RCBench can fully benefit from the advantages offered by RDMA-capable networks. We conduct extensive experiments to evaluate the scalability of mainstream concurrency control algorithms. The results show that by exploiting the capabilities of RDMA, concurrency control algorithms in RCBench can obtain 42X performance improvement, and transaction scalability can be achieved in RCBench.

Keywords concurrency control · distributed transaction · transaction scalability · RDMA

1 Introduction

The capability to support distributed transaction processing in database systems is indispensable for many mission-critical applications, such as e-banking and e-commerce. However, it is generally believed that distributed transaction processing over TCP/IP networks cannot scale [68]. That is, the increasing number of involved data nodes per transaction makes the system performance drop significantly. This phenomenon is referred to as *weak transaction scalability*. To show this phenomenon, we implement a two-phase locking concurrency control algorithm (a.k.a. 2PL) on distributed framework Deneva [31] open-sourced by MIT, and conduct an evaluation over YCSB benchmark. We plot the throughput by varying the number of accessed data nodes per transaction in Figure 1. The throughput of distributed transaction processing decreases by a factor of 75% when the number of accessed data nodes per transaction increases from 2 to 5.

The reasons that cause the weak transaction scalability are two-fold. On the one hand, communications from the coordinator to the participants increase heavily as more accessed data nodes per transaction are involved, making the system performance drop; on the other hand, coordinating more nodes in each transaction would introduce additional overhead, bottlenecked by the slowest one in the distributed transaction processing. Interestingly, to the best of our knowledge, none

Hongyao Zhao, Jingyao Li, Wei Lu, Qian Zhang, Wanqing Yang, Jiajia Zhong and Xiaoyong Du
the Key Laboratory of Data Engineering and Knowledge Engineering, Ministry of Education, China, and School of Information, Renmin University of China, China.
E-mail: {hongyaozhao, li-jingyao, lu-wei, zhangqianzq, wanqingyang, zhongjiajia, duyong}@ruc.edu.cn

Haixiang Li and Anqun Pan
Tencent Inc., China.
E-mail: {blueseali, aaronpan}@ruc.edu.cn

Meihui Zhang
Beijing Institute of Technology, China.
E-mail: meihui_zhang@bit.edu.cn

Fig. 1 The throughput drops sharply when the number of accessed data nodes per transaction increases.

of the existing works directly target solving weak transaction scalability. Instead, most of them target accelerating the performance of distributed transaction processing. For these works, they are divided into three categories. Approaches of the first category attempt to either eliminate distributed transactions or minimize their number as much as possible. For these proposals, various partitioning approaches and partition placement schemes [23,41,46,18,69,58,52,49,1,2] are carefully designed so that partitions involved in the same transaction locate in the same data node. Yet, static partitioning works only if the optimal data placement is known a priori and never changes, while dynamic partitioning often suffers from an expensive data migration overhead. As opposed to the first category, approaches of the second category directly optimize the execution logic of distributed transactions. To do this, variants of 2PC are carefully designed to reduce the number of communications. For instance, Early Prepare [51] eliminates the prepare phase of 2PC and hence one round-trip from the coordinator to the participants is reduced; besides, deterministic concurrency control algorithms [55,39,45,28,43,27,26,38] can completely eliminate 2PC, and hence two round-trips from the coordinator to the participants are reduced; Despite these optimizations, the master-worker architectures in these works are still bottlenecked by the high network latency and coordination overhead, which hinder the transaction scalability. To alleviate the network and coordination overhead in the second category, approaches [22,62,5,16,63,66] of the third category adopt the shared-memory architecture, in which nodes are connected via RDMA-capable networks, i.e., high-performance Infiniband networks with the remote direct memory access (a.k.a. RDMA) capability. As reported in [10], RDMA-capable networks provide relatively comparable latency to main memory, but take significantly lower latency than TCP/IP networks. Thus, under the shared-memory architecture, a worker is scheduled to execute transactions entirely without the coordination of the coordinators by reading/writing remote data items directly via low-latency RDMA verbs. This idea is similar to that in centralized systems, as opposed to dividing each distributed transaction into multiple sub-transactions in the master-worker architectures.

In this paper, we aim to build a unified transaction testbed over RDMA-capable networks with two requirements. Requirement R1: this testbed must take the full advantages of RDMA-capable networks. Requirement R1 helps verify whether re-implementations of concurrency control algorithms are transaction-scalable under the condition that all advantages of RDMA-capable networks are used. Requirement R2: it is generic to re-implement mainstream concurrency control algorithms in this testbed. Requirement R2 helps achieve convenient re-implementations and a fair comparison among them. Thus far, although quite a few works are proposed for RDMA-capable networks, they do not satisfy either requirement R1 or requirement R2 or both. For instance, the extensions of 2PL [5,16,63,66] and optimistic concurrency control (OCC) [22,62] mainly focuses on locking/unlocking data items and validation, respectively. However, these works lack generality (requirement R2) because applying any of them individually is not enough to re-implement other concurrency control algorithms, e.g., such as choosing proper versions of data items in multi-version concurrency control algorithms (MVCC); additionally, some implementations, e.g., DrTM-H [62], do not take full advantages of RDMA-capable networks, which may potentially affect the evaluation of transaction scalability.

To satisfy the above two requirements, we propose a unified transaction testbed called RCBench. To address requirement R1, we make a careful redesign of the data access methods for concurrency control algorithms completely using one-sided RDMA verbs, which take full advantages of RDMA-capable networks but come with the programming limitation of requiring prior knowledge of a data item's address before accessing it. To solve this problem, in RCBench, we carefully design a key-to-address index, with which, to access a data item, we first obtain its remote address based on its key, and then access the data item based on the remote address using one-sided RDMA verbs. Besides, existing optimizations for RDMA-capable networks, including coroutine, doorbell batching, outstanding requests, and passive ack are integrated into RCBench.

To satisfy requirement R2, we collect the metadata for each concurrency control algorithm and abstract six primitives. Every concurrency control algorithm can be conveniently re-implemented by invoking the primitives on its metadata, without directly touching the RDMA programming but enjoying all advantages of RDMA-capable networks. Because a concurrency control algorithm could be re-implemented using different set of primitives, or the same set of primitives but with varying numbers of RDMA verb invocations, we propose five optimization principles for the re-implementations that aim to achieve transaction scalability by minimizing the number of RDMA verb invocations. Note that, our proposed primitives differ from those designed in FaRM [21] which aim to speed up the message communication, while ours are used for manipulating remote metadata or data. Following these principles, we employ the primitives to re-implement multiple mainstream state-of-the-art concurrency control algorithms, including (1) widely-used protocols such as 2PL [29]: No-Wait [8], Wait-Die [47], Wound-Wait [47], T/O [47, 7,8], and MVCC [64]; (2) modern protocols such as Silo [59], Maat [30], and Cicada [37]; (3) a deterministic protocol Calvin [55].

To answer the questions that whether or not the re-implementations are transaction-scalable, and which re-implementations achieve the best performance, we conduct comprehensive experiments over the widely used benchmarks. We report our findings below.

- In RCBench, it is convenient to re-implement the concurrency control algorithms using our proposed six primitives that enjoy all benefits of RDMA-capable networks. Interestingly, some results show that our re-implementations can even achieve a better performance than some customized implementations under the same settings.
- All re-implementations, except for Calvin, demonstrate significant performance improvement (ranging from 18X to 42X) against their counterparts over TCP/IP networks. The degree of improvement closely relies on the number of primitive calls. Particularly, Silo is reported to perform the best in most cases.
- Among all optimizations, coroutine brings the greatest performance improvement (1.7X to 2.5X). Besides, the selection of appropriate lock types is also important. For example, using a single type of the exclusive lock instead of exclusive/shared locks in 2PL can significantly improve performance in moderate or low contention scenarios.
- Our experimental results show that transaction scalability can be achieved. We must emphasize this finding is rather important because in this way, it is unnecessary to do expensive data migration across data nodes in order to eliminate distributed transactions.

2 Preliminaries

In this section, we first review the distributed transaction processing and then discuss the background of RDMA techniques.

2.1 Distributed Transaction Processing

In shared-nothing database systems, data are horizontally partitioned, and data nodes are responsible for storing and accessing the partitions that are assigned to them. These systems adopt multi-coordinator system architecture to do distributed transaction processing, such as Percolator [42]. Specifically, each coordinator individually 1) accepts the transactions, 2) breaks the transaction into several sub-transactions and distributes sub-transactions to the appropriate data nodes, also known as participants, for execution, and 3) issues 2PC to coordinate the commit/abort of the transaction. In step 2), each data node is equipped with a database instance that manages the concurrent execution of the sub-transactions in this node. In step 3), once all sub-transactions decide to commit, the coordinator coordinates the commit of the transaction; otherwise, it coordinates the abort of the transaction.

2.2 RDMA

RDMA is a conceptual extension of direct memory access (DMA) technology and has become an increasingly popular technique to accelerate the performance of a system. It is capable of providing the following three properties. (1) **Zero-copy** property. Applications can perform data transfers without the involvement of the network software stack. Data are sent and received directly to the buffers without being copied between the network layers. (2) **Kernel bypass** property. Applications can perform data transfers directly from user space without kernel involvement. (3) **No CPU involvement** property. Applications can access remote memory without consuming any CPU cycles in the remote machine. Although RDMA is supported on both Ethernet and InfiniBand networks that provide the same common user APIs for programming, RDMA over InfiniBand networks provides much higher bandwidth and lower latency. Hence, in this paper, we focus on RDMA over InfiniBand networks.

It provides two categories of verbs for programming: (1) one-sided verbs, including READ, WRITE, WRITE With Immediate, and two atomic operations: FETCH-And-ADD (a.k.a. FAA) as well as COMPARE-And-SWAP (a.k.a. CAS), and (2) two-sided verbs, including

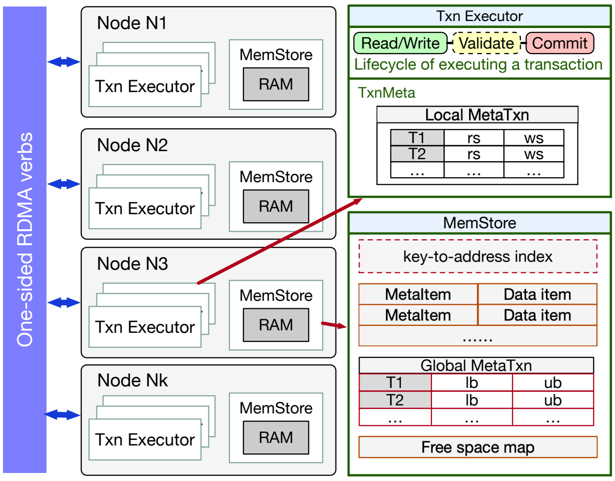


Fig. 2 An overview of RCBench

SEND and RECEIVE. Programming using one-sided verbs enjoys all three properties of RDMA. For example, the atomic RDMA CAS allows a machine to do compare-and-swap in the remote machine atomically without any intervention of the remote CPU. Nevertheless, programming using two-sided verbs can only have zero-copy and kernel-passing properties. That is, two-sided verbs still require CPU involvement of the remote machine. Therefore, a one-sided verb always seems more favorable than a two-sided verb.

Although one-sided verbs have all benefits of RDMA, in programming, we must know the memory address of the data to be read/written in the remote node a priori. That is, we may issue multiple one-sided verbs to obtain the memory address in the remote node and issue another verb to do the remote read/write using the remote address. On the contrary, we can read/write a remote data item by issuing a single two-sided verb without knowing its remote memory address. As reported in Section 7.3, we observe that 10–20 one-sided verb invocations are roughly equivalent to a single two-sided verb invocation, and system throughput decreases as the number of one-sided verb invocations increases. Therefore, in order to fully leverage the advantages of RDMA-capable networks, RCBench completely utilizes one-sided verb invocation to access remote data while reducing the number of one-sided verb invocations as much as possible.

3 Overview of RCBench

Table 1 Symbols and their meanings.

Symbol	Meaning
X	A data item
X^d	The data value of X
$X.PK$	The primary key of X
X^m	The ItemMeta of X
T	A transaction
$T.Tid$	ID of T
$T^l(T^g)$	Local (global) metadata of T
$T^l.rs$	The read set of T
$T^l.ws$	The write set of T
$T^g.lock$	The lock used to prevent concurrent writes on T^g
$X^m.lock$	The lock used to prevent concurrent writes on X^d and X^m

Figure 2 presents an overview of RCBench, which processes distributed transactions with the full benefits of RDMA-capable networks. We have released its source code on Github.[1] RCBench adopts the shared-memory architecture [10], in which data items are horizontally partitioned using a hash-based method. Each partition is assigned to one node and maintained in its main memory, and all operations on remote data items are executed via one-sided RDMA verbs. In RCBench, multiple clients generate transactions and send them to different servers in a round-robin fashion [31]. Each server acts as both a data node equipped with *MemStore* and a compute node equipped with multiple *Txn Executors*. For ease of illustration, Table 1 summarizes the notations used throughout the paper.

3.1 MemStore

MemStore is a pre-allocated, RDMA-registered memory that manages data items and provides auxiliary metadata to facilitate Txn executors in performing concurrency control. To facilitate the access of data items and metadata, we construct a key-to-address index on each node, where each index entry records the primary key and the main memory address of the corresponding data item residing in MemStore. Txn executors traverses the remote index to fetch data items using one-sided RDMA verbs. We will elaborate on the access of data items in Section 4.1.

We organize MemStore into several memory blocks in fixed-length, and maintain a bitmap $free\_space\_map$ to reflect the memory usage. In $free\_space\_map$, we use 1 bit to indicate the status of a memory block: 1 means the block is used, while 0 means it is free. When inserting or deleting a data item, we update the corresponding bit in $free\_space\_map$ atomically, which will be further discussed in Section 4.2.

Each data item X is stored in a fixed-length memory block, containing its corresponding metadata X^m and

[1] https://github.com/dbiir/RCBench

data value X^d. This allows remote Txn executors to access both X^m and X^d with a single one-sided RDMA verb. For metadata, we design the following two categories of metadata for performing concurrency control.

- **MetaItem** X^m. X^m represents the metadata corresponding to each data item X. For example, in 2PL, X^m includes the ID of a running transaction that has ever written X, or the ID list of running transactions that have ever read X, serving as the exclusive/shared lock identifiers. In the real implementation, X^m is set as a fixed-length data structure. By doing this, our access method guarantees that X^m and T^g can be fetched using a single one-sided RDMA verb if we have the address of X. Suppose in 2PL, we store the ID list of transactions in X^m. Because we cannot know how many transactions read each data item X a priori, we create X^m with a very large list to store them, which leads to a prohibitively expensive space overhead. As a comprise, we set a proper size of the list. When a transaction T attempts to acquire a shared lock on X, and the ID list of X^m is full, we simply abort T to ensure correctness.
- **MetaTxn** T^l/T^g. MetaTxn denotes metadata maintained by a transaction T, including local metadata T^l and global metadata T^g. Local metadata T^l can only be manipulated by the local Txn executors, so we store T^l in the local memory of Txn executors. In contrast, global metadata T^g can be accessed by remote Txn executors. For example, in 2PL, T^l maintains the transaction's read/write sets, and global MetaTxn T^g contains the transaction status, including running, aborted, or committed. We manage T^g similarly to data items and allocate fixed-length memory for each T^g, which will be presented in detail in Section 4.3.

3.2 Txn Executor

Every Txn executor in our framework works like its counterpart in the centralized system except that the former is enriched with the capability to directly access the memory of a remote node. After receiving a transaction T, a Txn executor processes T through two or three phases: 1) read/write phase, 2) validation phase (if any), and 3) commit phase. In the read/write phase, the Txn executor checks the condition, e.g., acquire a lock using 2PL, to read a data item X, or/and possibly write X. Before this, it is necessary to fetch the address of X in the local/remote node through the key-to-address index. In the validation phase (if any), the Txn executor examines the conflicts between T and other concurrent transactions and determines whether or not T can commit or abort. If T aborts, any modifications by T need to be rollbacked; otherwise, any modifications by T need to be persisted in Memstores.

Although it is general to abstract the execution of each transaction through two or three phases, concurrency control algorithms could have different logic to ensure transaction correctness. Obviously, individual re-implementations of them are inefficient for development, and more importantly, we attempt to make a fair comparison over these re-implementations. For these reasons, we first collect the metadata for each concurrency control algorithm and abstract six primitives. Every concurrency control algorithm can be conveniently re-implemented by invoking the primitives on its metadata, without directly touching the RDMA programming, but enjoying all advantages of RDMA-capable networks. Because a concurrency control algorithm can be re-implemented with a different set of primitives, or with the same set of primitives but having a significantly different number of RDMA verb invocations, aiming to achieve transaction scalability, we propose five optimization principles for the re-implementations that are able to reduce the number of RDMA verb invocations.

To further improve performance, we provide two transaction processing modes for Txn executor, including the thread-to-transaction mode and coroutine-to-transaction mode. In the first mode, a thread is created for one executor to execute transactions sequentially; in the second mode, a thread is composed of multiple coroutines, each of which is created for one executor to execute transactions sequentially. By using finer-grained scheduling in the second mode, when a coroutine is blocked, the transaction execution can be switched to another coroutine of the same thread. The scheduling overhead of coroutines is significantly smaller than that of threads, and hence coroutine-to-transaction mode can achieve better performance.

4 The Access Method

In this section, we present the access and maintainence method of remote data items and global metadata completely using one-sided RDMA verbs.

4.1 The Access to Remote Data Items

As mentioned in Section 3, we develop an RDMA-friendly key-to-address hash index IDX to manage data items on each node. In general, reading or writing a data item is accomplished using two one-sided RDMA verbs. When accessing a data item, given its primary key, we first retrieve the corresponding index entry using a one-sided RDMA verb, and then obtain the data item based on the address stored in the index entry with another one-sided RDMA verb. However, additional one-sided RDMA verbs for fetching an index entry are required when hash collisions happen, resulting in increased network overhead. To address this issue, we adopt the empirical methods from [40] and implement the hash index IDX as an n-way Cuckoo hash table. With this hash scheme, we apply n orthogonal hash functions, denoted as $hash_k$ $(k = 1, 2, ..., n)$, to assign n locations for a primary key, indicating that every key is either at one of n possible locations or absent. By default, we set $n = 3$, referring to the optimal setting indicated in [40]. By iteratively examining these n possible locations, we guarantee that an index entry can be located using $\leq n$ single one-side verbs. We elaborate on how to fetch an index entry through IDX in detail:

In our design, we store an index entry as a triple $\hat{X}$ of $\langle X.addr, X.size, X.PK\rangle$ occupying 24 bytes, where $X.addr$, $X.size$, and $X.PK$ are the memory address, size, and primary key of the data item X, respectively. Besides, each index entry maintains a *lock*, occupying 8 bytes, to prevent concurrent manipulation on IDX. By so doing, we formulate the function *getRtItemAddr* to fetch the appropriate index entry $\hat{X}$ of a given key PK below: (1) we calculate a location using $hash_k(X.PK)$ and issue an RDMA READ to fetch its corresponding $\hat{X}$; (2) we then return $\hat{X}$ if $\hat{X}.X.PK = PK$ and $\hat{X}$ is not locked by another transaction. If $\hat{X}$ has been locked by another transaction, we wait until the lock is released. If $\hat{X}.X.PK \neq PK$, we set $k = k + 1$ and re-execute (1)(2) until either finding a correct $\hat{X}$ with $\hat{X}.X.PK = PK$, or retrying until $k = n - 1$.

To support range queries, we adopt the approach proposed in [32]. Specifically, we horizontally partition index entries into several hash indexes based on the prefix of the stored primary key. When querying data items with primary key ranges from PK_i to PK_j, we traverse hash indexes that exhibit intersection with the requested range to obtain required index entries. To prevent phantom read anomalies, we maintain an extra lock meta $IDX.lock$ for each hash index. Before/After the range query, shared locks are acquired/released on involved hash indexes.

4.2 The Manipulation of Remote Data Items

We elaborate on the manipulations of remote data items in terms of insert and delete. Based on the design of IDX, inserting a new index entry $\langle X.addr, X.size, PK\rangle$ of X requires the following procedures. (1) First, n RDMA CAS invocations are issued to concurrently lock n potential locations with addresses that are calculated by $hash_k(X.PK)$ for $k = 1, 2, ..., n$. In case of any failure, additional RDMA CAS are required to retry locking until success. (2) Then, we issue n concurrent RDMA READ invocations to obtain these locations, and examine them locally. (3) If any free location is available, an RDMA WRITE invocation is used to write the new index entry into the corresponding location. (4) Otherwise, insertion with preemption is executed by invoking an RDMA WRITE to kick out and replace one of the existing locations with the new one. (5) Finally, we release all prior locks with n RDMA CAS invocations. Note that, if preemption exists in this procedure, similar steps are repeated to insert the kicked-out entry. To prevent successive kicks, the hash table will be resized to accommodate more data if the number of kicks reaches a pre-defined limit [40].

Before insertion of the index entry, a free memory space needs to be found for the data item X in the remote MemStore. To do this, we issue an RDMA READ to fetch $free\_space\_map$, identify one of the available memory spaces based on it, and invoke an RDMA CAS to reset the corresponding bit to 1 before writing X.

When deleting a data item X, we (1) issue n RDMA READ to obtain index entries with addresses calculated by $hash_k(X.PK)$ for $k = 1, 2, ..., n$. (2) use an RDMA CAS to lock the target entry $\hat{X}$, and invoke another RDMA READ afterward to confirm that $\hat{X}$ has not been modified by any other transaction. (3) clean $\hat{X}$ from IDX with an RDMA WRITE, and release the memory space of X by issuing an RDMA CAS to update the corresponding bit in $free\_space\_map$ from 1 to 0.

In contrast to the shared locks for range queries in Section 4.1, we acquire/release exclusive locks on hash indexes before/after insert or delete operations to prevent the phantom read. We will present the implementation of shared and exclusive locks in Section 6.2.3.

4.3 The Access to Global Metadata

We manage the global metadata T^g in the same manner as that of data items, which occupies a fixed-length Θ of memory space in MemStore. We use an n-way cuckoo hash table to store the index entries of global MetaTxn T^g. With this hash table, the access, insert and delete procedures for a given T^g are basically the same as that of data items described in Section 4.1 and Section 4.2. Each index entry is stored as a tuple $\hat{T}$ of $\langle T^g.addr, T.Tid\rangle$, where $T.Tid$ is taken as an integer.

Besides, each index entry maintains a *lock*, occupying 8 bytes, to prevent concurrent manipulations. For reference, we name the function used to obtain the appropriate index entry $\hat{T}$ of a given Tid as $getRtTgAddr$, and omit its implementation as it is self-explanatory.

5 Concurrency Control Primitives

In this section, we first formulate the operation logic of concurrency control algorithms in centralized database systems, then abstract six primitives that take full benefits of RDMA-capable networks, and finally extend the operations based on the primitives to facilitate the reimplementations of concurrency control algorithms.

5.1 Operation Abstraction in Centralized Systems

A transaction can be modeled as a sequence of read/write operations, ended with a commit/abort operation. In the centralized database system, upon any read/write from/to data item X^d, the concurrency control algorithms need to examine whether the transaction has the qualification to read/write X^d by acquiring a lock on X^d in 2PL, or examine the timestamp on X^d in T/O, or others. To verify whether a transaction can commit or needs to abort, the concurrency control algorithms might need to do the validation, by either checking the conflict between its read set and the write set of concurrent transactions, or adjusting its timestamp interval based on its concurrent transactions, or others. To do the commit/abort, the concurrency control algorithms might need to release the locks, and persist the write set of the transaction. For reference, we list the operations that are necessary for some classic concurrency control algorithms in Table 2.

The execution of all operations in Table 2 can be formulated into three steps, shown in Algorithm 1: (1) fetch data item X (line 1), (2) perform the operation logic based on X (line 2), and (3) update X if necessary (line 3). We take a lock request on a data item X in lock-based algorithms for an example to illustrate these three steps. To acquire a lock on X, we (1) first fetch X^m, (2) then examine whether the lock meta $X^m.lock$ in X^m has been modified by other transactions; (3) and finally update X^m by setting $X^m.lock$ to indicate that X has been locked by it if $X^m.lock$ has not been modified by other transactions.

5.2 Primitive Abstraction in RDMA-capable Networks

To extend the operations of concurrency control algorithms in RDMA-capable networks, and make the underlying RDMA programming transparent to developers, we abstract six primitives, which are $Read^D$, $Write^D$, $Atomic^D$ for data items and $Read^T$, $Write^T$, $Atomic^T$ for transaction's meta items.

Algorithm 1: Operation logic abstraction in centralized database systems

```
1 Fetch $X^d$ and/or $X^m$;
2 Perform operation logic based on $X^d$ and/or $X^m$;
3 Update $X^d$ and/or $X^m$ if necessary.
```

• $Read^D$ (Primitive 1) is used to read remote X. In each node, we maintain the address of IDX of every data node. By taking PK as input, $Read^D$ fetches $\hat{X}$ by issuing $getRtItemAddr$ function (line 1). If $\hat{X} = NULL$, $Read^D$ fails and returns $NULL$ since $\hat{X}$ does not exist (line 2); otherwise, $Read^D$ issues another RDMA READ to read and return X via $\hat{X}$ (line 3).

Primitive 1: $Read^D(PK)$

```
1 $\hat{X} \leftarrow getRtItemAddr(PK)$;
2 if $\hat{X} = NULL$ then return $NULL$;
3 return $RDMA\_READ(\hat{X}.X.addr, \hat{X}.X.size)$
```

• $Write^D$ (Primitive 2) is designed to overwrite remote X. Similarly, $Write^D$ issues $getRtItemAddr$ function to fetch $\hat{X}$ (lines 1–2). If $\hat{X}$ exists, then $Write^D$ issues another RDMA WRITE that writes the new value $newV$ to X via $\hat{X}$ (line 3).

Primitive 2: $Write^D(PK, newV)$

```
1 $\hat{X} \leftarrow getRtItemAddr(PK)$;
2 if $\hat{X} = NULL$ then return false;
3 $RDMA\_WRITE(\hat{X}.X.addr, newV, \hat{X}.X.size)$;
4 return true;
```

• $Atomic^D$ (Primitive 3) is designed to conditionally update an item (specified by input parameter $meta$) of X^m, e.g., the lock of X ($X^m.lock$ with $meta = MT\_LOCK$) or the read timestamp of X ($X^m.rts$ with $meta = MT\_RTS$), with the atomicity guarantee. Given PK, $Atomic^D$ issues $getRtItemAddr$ to fetch $\hat{X}$ (line 1). If $\hat{X}$ exists, $Atomic^D$ calculates the remote address $X.meta\_addr$ of $meta$ by adding $meta$'s offset in X^m to $\hat{X}.X.addr$ (line 3). and issues RDMA CAS to conditionally update $meta$ by new value $newV$ with atomicity guarantee (line 4).

Table 2 Operations of classic algorithms. Each class of algorithms includes operations involving marking with a ✓.

Phase	Operations	Lock-based algorithms	OCC algorithms	Timestamp-based algorithms	MVCC-based algorithms
Read/Write	Read	✓	✓	✓	✓
	Write	✓	✓	✓	✓
	Lock	✓			
	Timestamp-Examination			✓	
	Version-Retrieval				✓
Validation	Read/Write-Set-Validation		✓		
	Timestamp-Interval-Adjust		✓		
	Lock		✓		
Commit	Persist-Data	✓	✓	✓	✓
	Roll-Back-Modification	✓	✓	✓	✓
	Unlock	✓	✓		

Primitive 3: $Atomic^D(PK, meta, oldV, newV)$

```
1 $\hat{X} \leftarrow getRtItemAddr(PK)$;
2 if $\hat{X} = NULL$ then return $false$;
3 $X.meta\_addr \leftarrow \hat{X}.X.addr + meta.offset$;
4 $t \leftarrow RDMA\_CAS(X.meta\_addr, oldV, newV)$;
5 return $t = oldV$;
```

•$Read^T$ (Primitive 4) is designed to read remote T^g. Taking Tid as the input, $Read^T$ issues $getRtTgAddr$ function to fetch $\hat{T}$ (line 1). If $\hat{T} = NULL$, $Read^D$ fails and returns $NULL$ since $\hat{T}$ does not exist (line 2); otherwise, $Read^T$ issues an RDMA READ to read T^g via $\hat{T}.T^g.addr$ and Θ, the length of data structure T^g, and return T^g (line 3).

Primitive 4: $Read^T(Tid)$

```
1 $\hat{T} \leftarrow getRtTgAddr(Tid)$ ;
2 if $\hat{T} = NULL$ then return $NULL$;
3 return $RDMA\_READ(\hat{T}.T^g.addr, \Theta)$
```

• $Write^T$ (Primitive 5) is designed to write remote T^g. $Write^T$ first issues $getRtTgAddr$ function to fetch $\hat{T}$ (lines 1–2), and issues an RDMA WRITE to overwrite T^g with $newV$ (line 3).

Primitive 5: $Write^T(Tid, newV)$

```
1 $\hat{T} \leftarrow getRtTgAddr(Tid)$ ;
2 if $\hat{T} = NULL$ then return $false$;
3 $RDMA\_WRITE(\hat{T}.T^g.addr, newV, \Theta)$;
4 return $true$;
```

•$Atomic^T$ (Primitive 6) is designed to conditionally update an item (specified by the input parameter $meta$) of T^g, e.g., $T^g.lock$, $T^g.st$ with atomicity guarantee. $Atomic^T$ fetches $\hat{T}$ by issuing function $getRtTgAddr$ (lines 1–2), calculates $T^g.meta\_addr$ of $meta$ by adding $meta$'s offset in T^g to $\hat{T}.T^g.addr$ locally (line 3) and issues an RDMA CAS to conditionally update $meta$ by new value $newV$ with atomicity guarantee (line 4).

Primitive 6: $Atomic^T(Tid, meta, oldV, newV)$

```
1 $\hat{T} \leftarrow getRtTgAddr(Tid)$ ;
2 if $\hat{T} = NULL$ then return $false$;
3 $T^g.meta\_addr \leftarrow \hat{T}.T^g.addr + meta.offset$;
4 $t \leftarrow RDMA\_CAS(T^g.meta\_addr, oldV, newV)$;
5 return $t = oldV$;
```

5.3 Operation Extension in RDMA-capable Networks

With the above six primitives that take full benefits of RDMA-capable networks, we extend the operations of concurrency control algorithms in centralized database systems to RDMA-capable networks. As for the access to remote data items, we propose RDMA-BasicD shown in Algorithm 2 as the extension. RDMA-BasicD takes the primary key PK of a data item and current transaction T as the input. It (1) first issues $Read^D$ to read remote data items (line 4), (2) then performs local logic based on X (line 7), and (3) finally modifies X using $Write^D$(line 9). For example, to acquire a lock on remote data item X in lock-based algorithms, we first use $Read^D$ to fetch X^m, then locally examine whether the lock meta $X^m.lock$ has been modified by other transactions, and finally update X^m with $X^m.lock$ using $Write^D$ to indicate that X has been locked if $X^m.lock$ has not been modified by other transactions. In our implementation, we acquire a latch to ensure the atomicity of the three steps using $Atomic^D$. As shown in Algorithm 2, we issue an $Atomic^D$ to acquire the latch

Algorithm 2: Operation extension in RDMA-capable networks

```
1  Function RDMA-BasicD($PK, T$):
2      $r \leftarrow Atomic^D(PK, MT\_LATCH, 0, T.Tid)$;
3      if $\neg r$ then $Abort\ T$;
4      $X \leftarrow Read^D(PK)$;
5      if $X = NULL$ then
6          $Atomic^D(PK, MT\_LATCH, T.Tid, 0)$;
           $Abort\ T$;
7      Perform local logic based on $X$;
8      $\hat{X}^m.latch \leftarrow 0$;
9      $Write^D(PK, X)$;
10 Function RDMA-BasicT($Tid_i, T$):
11     $r \leftarrow Atomic^T(Tid_i, MT\_LATCH, 0, T.Tid)$;
12     if $\neg r$ then $Abort\ T$;
13     $T_i^g \leftarrow Read^T(Tid_i)$;
14     if $T_i^g = NULL$ then
15         $Atomic^T(Tid_i, MT\_LATCH, T.Tid, 0)$;
           $Abort\ T$;
16     Perform local logic based on $T_i^g$;
17     $\hat{X}^m.latch \leftarrow 0$;
18     $Write^T(Tid_i, T_i^g)$;
```

of data item X (lines 2,3), and set $X^m.latch$ to 0 to release this latch (lines 8,9). As for the access to remote global transaction metadata T^g, we further propose RDMA-BasicT shown in Algorithm 2 as the extension. We omit the details of RDMA-BasicT which follow the same logic as RDMA-BasicD.

Overall, RDMA-BasicD and RDMA-BasicT can be used to implement all remote operations of mainstream concurrency control algorithms. We take lock-based algorithms as an example, all their operations can be implemented using the same three steps outlined in the lock acquisition logic. In conclusion, RDMA-BasicD can be used to implement timestamp-examination, version-retrieval, and read/write-set validation operations, and RDMA-BasicT can be used to implement timestamp-interval-adjust operations, covering all the operations mentioned in Table 2.

5.3.1 Re-implementations using RDMA-BasicD only

No-Wait [8] is a variant of 2PL concurrency control algorithm. For any data item X, it always tries to acquire a certain type of lock on X before any read or write on it and aborts immediately in case of locking failures to avoid deadlocks. Specifically, in the read/write phase of No Wait, for each read of X, a shared lock is acquired on X, while for each write of X, an exclusive lock of X is acquired. In the Commit phase, we update the values of the data items that need to be modified, and release all the locks.

Algorithm 3: RDMA-No-Wait using RDMA-BasicD

```
1  Function Read($PK, T$):
2      if $\neg\ Atomic^D(PK, MT\_LATCH, 0, T.Tid)$ then
         $Abort\ T$;
3      $X \leftarrow Read^D(PK)$;
4      if $X^m.lock\_type == EL$ then
5          $Atomic^D(PK, MT\_LATCH, T.Tid, 0)$;
6          $Abort\ T$;
7      $X^m.lock\_type \leftarrow SH$;
8      $X^m.lock\_list \leftarrow X^m.lock\_list \cup T.Tid$;
9      $X^m.latch \leftarrow 0$;
10     $Write^D$(X.PK, $X$);
11     $T^l.rs \leftarrow X \cup T^l.rs$;
12 Function Write($PK, newV, T$):
13     if $\neg\ Atomic^D(PK, MT\_LATCH, 0, T.Tid)$ then
         $Abort\ T$;
14     $X \leftarrow Read^D(PK)$;
15     if $X^m.lock\_type == EL\ or\ SH$ then
16         $Atomic^D(PK, MT\_LATCH, T.Tid, 0)$;
17         $Abort\ T$;
18     $X^m.lock\_type \leftarrow EL$;
19     $X^m.lock\_list \leftarrow X^m.lock\_list \cup T.Tid$;
20     $X^m.latch \leftarrow 0$;
21     $Write^D$(X.PK, $X$);
22     $T^l.ws \leftarrow \langle X, newV \rangle \cup T^l.ws$;
23 Function Commit($T$):
24     foreach $X \in T^l.rs$ do
25         while $\neg Atomic^D$(X.PK, $MT\_LATCH$,
             0,$T.Tid$);
26         $X \leftarrow Read^D(PK)$;
27         $X^m.lock\_type \leftarrow EL$;
28         $X^m.lock\_list \leftarrow X^m.lock\_list - T.Tid$;
29         if $X^m.lock\_list = \varnothing$ then
30             $X^m.lock\_type \leftarrow 0$;
31         $X^m.latch \leftarrow 0$;
32         $Write^D$(X.PK, $X$);
33     foreach $\langle X, newV \rangle \in T^l.ws$ do
34         while $\neg Atomic^D$(X.PK, $MT\_LATCH$,
             0,$T.Tid$);
35         $X \leftarrow Read^D(PK)$;
36         $X^m.lock\_type \leftarrow EL$;
37         $X^m.lock\_list \leftarrow X^m.lock\_list - T.Tid$;
38         $X^m.lock\_type \leftarrow 0$;
39         $X^d \leftarrow newV$;
40         $X^m.latch \leftarrow 0$;
41         $Write^D$(X.PK, $X$);
```

To boost No-Wait, we use RDMA-BasicD to implement the logic that (1) acquire the remote exclusive lock and do remote write, (2) acquire the remote shared lock and do remote read, and (3) release remote exclusive/shared locks. X^m includes (1) latch metadata *latch* to prevent concurrent modifications on X, (2) $lock_t ype$ to indicate the current lock type on X (0 for no lock or NL, 1 for shared lock or SL, and 2 for exclusive lock or EL), and (3) the list $X^m.lock_l ist$, where each item stores the ID of the transaction that has a lock on X. Additionally, MetaTxn T^l maintains the read set $T^l.rs$ and write set $T^l.ws$ of transaction T. The key functions of the re-implementation of RDMA-No-Wait are exhibited in Algorithm 3.

Function *Read* is used to acquire a shared lock on X and read the data item X. The process begins with the issuance of an $Atomic^D$ to acquire the latch on X (line 2), then uses $Read^D$ to fetch X (line 3), and checks whether X has been granted an exclusive lock to another transaction (lines 4). If X has been granted an exclusive lock to another transaction, the function releases the latch and aborts T (lines 4–6). If not, the function adds T into $X^m.lock_list$, updates $X^m.lock_type$ to SH, sets $X^m.latch$ to 0, and invokes $Write^D$ to overwrite the remote X (lines 7–10). Finally, the function records X into the read set $T^l.rs$ of T (line 11).

Function *Write* is used to acquire an exclusive lock on X and perform a remote write following the same steps as function *Read*. We first acquire the latch on X (line 13), then use $Read^D$ to fetch X (line 14), and check whether X has been granted a lock to another transaction. If so, we release the latch and abort T (lines 15–17); otherwise, we add T into $X^m.lock_list$, update $X^m.lock_type$ to EL, set $X^m.latch$ to 0, and invoke $Write^D$ to overwrite the remote X (lines 18–21). Finally, we add this modified X into the write set $T^l.ws$ (lines 22–23).

Function *Commit* is used to release the lock acquired before and persist the updated values. For each data item X in $T^l.rs$, we first acquire the latch on X using $Atomic^D$ to ensure that X is not being modified by other transactions (line 24). Then, we remove T from $X^m.lock\_list$ and check if there are no more transactions holding a lock on X. If so, we set $X^m.lock\_type$ to NL (lines 29–30). Finally, we release the latch on X by invoking $Write^D$ (lines 31–32). For each data item X in $T^l.ws$, we first acquire the latch on X using $Atomic^D$ to ensure that X is not being modified by other transactions (line 34). Then, we remove T from $X^m.lock_list$, set $X^m.lock_type$ to NL, update the value of X to the new value $newV$, and set $X^m.latch$ to 0 by invoking $Write^D$ (lines 35–40).

6 Design Principles and Re-Implementations of Concurrency Control Algorithms

6.1 Optimization Principles

As discussed, remote operations of various algorithms can be implemented with RDMA-BasicD and RDMA-BasicT conveniently. However, since RDMA-BasicD and RDMA-BasicT are general operations feasible for implementing all the concurrency control algorithms, further optimizations are required to fit the characteristics of different algorithms. We introduce five optimization principles for RDMA-BasicD and RDMA-BasicT, which either reduce the number of one-sided verb invocations, or eliminate explicit latch acquisition. By deeply customizing the re-implementations of concurrency control algorithms using these principles, we fully leverage the advantages of RDMA-capable networks to achieve transaction scalability.

First, as discussed in Section 2.2, **reducing the number of one-sided verb invocations can improve performance**. As shown in Algorithm 2, both BasicD and BasicT require three primitives to complete a remote operation. However, depending on the characteristics of the algorithm, we find that not all three primitives are necessary. For example, the lock acquisition operation in lock-based algorithms can be implemented using a single $Atomic^D$. By eliminating primitives in RDMA-BasicD based on the requirements of various algorithms, we have designed four optimization principles: One-Cas, One-Write, One-Read, which uses only one $Atomic^D$, $Write^D$, $Read^D$ in RDMA-BasicD; and Read-Cas, which uses a $Read^D$ and a $Atomic^D$ in RDMA-BasicD. These four principles are also applicable to RDMA-BasicT. We provide an example of One-Cas, Read-Cas, and One-Write in Section 6.2.3, and an example of One-Read in Section **??**, respectively.

- **One-Cas.** If an operation only modifies metadata with a maximum of 8 bytes and knows the original and new values in advance, we can use a single $Atomic^D$ ($Atomic^T$) to modify this metadata.
- **One-Write.** If a transaction T has acquired a lock or latch for the target metadata, T can directly use a $Write^D$ ($Write^T$) to write back the metadata modified.
- **One-Read.** If there is no need to modify the remote metadata, a single $Read^D$ ($Read^T$) would be enough to read it.
- **Read-Cas.** If an operation only modifies the metadata with a maximum of 8 bytes, but its new value needs to be calculated against the old value, we can issue one $Read^D$ ($Read^T$) to read back the metadata and another $Atomic^D$ ($Atomic^T$) to update it.

Second, **explicit latch acquisition may reduce concurrency when accessing data items, incurring higher abort rate**. For example, the read operation in T/O algorithm can be performed without the explicit latch to improve concurrency, as described in Section 6.2.2. To eliminate latch acquisition, we design the Double-Read principle as follows.

- **Double-Read.** Double-Read is an alternative for atomicity guarantee other than explicit latches. Specifically, it issues two $Read^D$($Read^T$) before and after a potential modification of the target metadata using an $Atomic^D$ ($Atomic^T$). If the content re-read is different from the first one in an unexpected way, a concurrent modification may have occurred.

6.2 Re-implementations

Based on our proposed primitives and optimization principles, we manage to re-implement mainstream algorithms by minimizing the number of one-sided verbs and reducing explicit latch acquisition. Now we present how to re-implement concurrency control algorithms completely based on the proposed six primitives and five optimization principles. The operations that use the principles will be highlighted in pink.

6.2.1 Lock-based Algorithms

In Section 5.3.1, we presented an implementation of No-Wait using RDMA-BasicD only. With the application of the five optimization principles, we can further optimize RDMA-No-Wait according to its characteristic.

In No-Wait, we only need to check whether a data item X is locked and how many transactions have locked it, without considering which transaction holds the lock. Thus, we make special lock metadata, $X^m.lock$, in the metadata of each data item X^m. The $X^m.lock$ is a 64-bit value, where the least significant bit is used to store the *lock_type* (0 for shared lock or SL, and 1 for exclusive lock or EL), and the remaining 63 bits store the number of shared locks held on X (i.e., *num_of_locks*). By doing this, we can acquire an exclusive lock on X by updating $X^m.lock$ from 0 to EL, satisfying the requirements of the One-Cas. And we can acquire a shared lock by adding 1 into $X^m.lock.num\_of\_locks$, satisfying the requirements of the Read-Cas. As transaction T has already acquired locks for data items when it enters the commit phase, the One-Write principle is applicable for the commit phase. Besides, MetaTxn T^l maintains the read set $T^l.rs$ and write set $T^l.ws$ of transaction T. Algorithm 4 exhibits the key functions of the re-implementation RDMA-No-Wait.

Function *AcqIMExcLock* and *RlsIMExcLock* are used to acquire/release an exclusive lock on X, which apply One-Cas principle. We acquire/release the exclusive lock on X by modifying the remote $X^m.lock$ from $0/EL$ to $EL/0$ through only one $Atomic^D$(lines 2, 5).

Function *AcqIMShLock* uses the Read-Cas optimization principle to acquire a shared lock. By taking its primary key PK as the input, we first issue a $Read^D$ to obtain $X^m.lock$ (line 7) and check whether there is an exclusive lock on X locally (line 8). If there is, it fails to acquire the lock; otherwise, we make a local copy $newL$ of $X^m.lock$, and update $newL$ by recording a new shared lock on X (lines 11–12); we then issue an $Atomic^D$ to update $X^m.lock$ with $newL$ to declare that a new shared lock on X is granted (line 13). Note that, $Atomic^D$ compares the remote $X^m.lock$ with local $X^m.lock$ and modifies remote $X^m.lock$ to $newL$ only if they are the same; that is, if any changes to X have been made between *ReadDI* (line 7) and $Atomic^D$ (line 13), the remote write would fail, and the returned value r is set to be false (line 14). Following the reverse logic of *AcqIMShLock*, function *RlsIMShLock* also uses this principle to release remote shared locks (lines 15–20).

Function *Commit* is used to commit transaction T and uses the One-Write principle. For each data item read before, we sequentially release the shared locks that T has acquired (lines 30–31) using *RlsIMShLock*. For each data item X in the write set $T^l.ws$, we issue *PersistData* to write the new data into remote nodes (lines 32–33).

Functions *Read* and *Write* are used to do remote reads and writes, respectively. For *Read* or *Write*, we first try to acquire the lock on X with $X.PK = PK$, then read the remote X if the lock acquisition success, and add X to the local read/write set $T^l.rs/T^l.ws$. Note, for *Write*, we need to update X^d locally before adding it into the local write set $T^l.ws$ (line 27).

Discussion. As shown in Algorithm 4, it requires at least two primitive calls ($Read^D$ and $Atomic^D$) to acquire/release a shared lock while it requires only one primitive call ($Atomic^D$) to acquire/release an exclusive lock. Based on this observation, for the low-conflict application scenarios where reads/writes on the same data item rarely occur, using a single exclusive lock instead of exclusive/shared locks could potentially bring performance benefits by reducing extra remote primitive calls. To verify the benefits of using the single exclusive locking mechanism, we make an extensive experimental evaluation in Section 7.6, and the result shows that single exclusive locking can outperform exclusive/shared locking by a factor of 1.3X in the low-conflict application scenarios. *For this reason, in this paper, we adopt the single exclusive locking mechanism instead of exclusive/shared locking mechanism by default.* In the real implementation, we collectively use *AcqIMExcLock*/*RlsIMExcLock* instead of *AcqIMShLock* / *RlsIMShLock* in Algorithm 4. For illustration purposes,

Algorithm 4: RDMA-No-Wait

```
1  Function AcqIMExcLock($PK$):
2      $r \leftarrow Atomic^D(PK, MT\_LOCK, 0, EL)$;
3      return $r$;
4  Function RlsIMExcLock($PK$):
5      $Atomic^D(PK, MT\_LOCK, EL, 0)$);
6  Function AcqIMShLock($PK$):
7      $X \leftarrow Read^D(PK)$;
8      if $X^m.lock\ .lock\_type = EL$ then
9          return $false$;
10     else
11         $newL \leftarrow X^m.lock$;
12         $newL.num\_of\_locks{+}{+}$;
13         $r \leftarrow Atomic^D(PK,\ MT\_LOCK,\ X^m.lock,\ newL)$;
14         return $r$;
15 Function RlsIMShLock($PK$):
16     $X \leftarrow Read^D(PK)$;
17     $newL \leftarrow X^m.lock$;
18     $newL.num\_of\_locks - -$;
19     $r \leftarrow Atomic^D(PK, MT\_LOCK, X^m.lock\ , newL)$;
20     if $\neg r$ then goto line 11;
21 Function Read($PK, T$):
22     if $\neg AcqIMShLock(PK)$ then $Abort\ T$;
23     $X \leftarrow Read^D(PK)$;
24     $T^l.rs \leftarrow \{X\} \cup T^l.rs$;
25 Function Write($PK, newV, T$):
26     if $\neg$ $AcqIMExcLock(PK)$ then $Abort\ T$;
27     $X \leftarrow Read^D(PK)$; $X^d \leftarrow newV$;
28     $T^l.ws \leftarrow \{X\} \cup T^l.ws$;
29 Function Commit($T$):
30     foreach $X \in T^l.rs$ do
31         $RlsIMShLock(X.PK\ )$;
32     foreach $X \in T^l.ws$ do
33         $X^m.lock \leftarrow 0$; $Write^D$(X.PK, $X$);
```

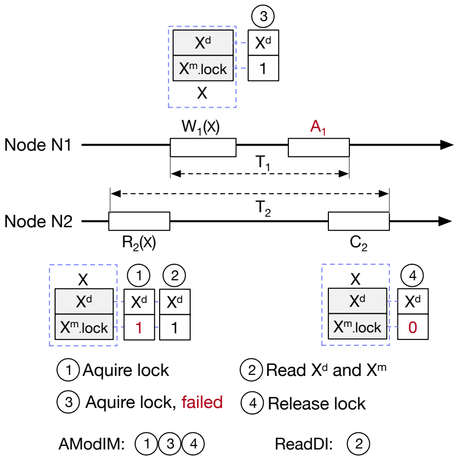


Fig. 3 An example of RDMA-No-Wait.

Algorithm 5: RDMA-Wound-Wait

```
Function LockWithWound($PK, T$):
    while $\neg$ $AcqIMExcLock(PK)$ do
        $X \leftarrow Read^D(PK)$;
        $\langle\ \overline{T}^l.bts\ , \overline{T}.Tid\ \rangle \leftarrow X^m.pL$;
        if $T^l.bts < \overline{T}^l.bts$ then
            $Atomic^T(\overline{T}.Tid, MT\_ST, RN, AB)$;
    $X \leftarrow Read^D(PK)$;
    $X^m.pL \leftarrow \langle\ T^l.bts\ , T.Tid\ \rangle$;
    $Write^D(PK, X)$;
    return $X$;
Function Commit($T$):
    if $\neg Atomic^T\ (\overline{T}.Tid, MT\_ST, RN, CM)$ then $Abort\ T$;
    foreach $X \in T^l.ws$ do
        $X^m.lock \leftarrow 0$; $Write^D(X.PK\ , X)$;
    foreach $X \in T^l.rs$ do
        $RlsIMExcLock(X.PK)$;
```

we show how the single exclusive locking mechanism works in the below example.

We consider two concurrent transactions T_1 and T_2 in Figure 3. T_2 first reads a data item X (denoted as $R_2(X)$), and then T_1 modifies X (denoted as $W_1(X)$). Suppose T_1 and X are located on the same node N_1, while transaction T_2 is located on another node N_2. Initially, there is no lock on X. Thus, for $R_2(X)$, T_2 acquires an exclusive lock on X by setting $X^m.lock$ from 0 to 1 through an $Atomic^D$ call (line 22, step ①); T_2 then issues a $Read^D$ call to read X from node N_1 to local node N_2 (line 23, step ②), and add X into $T_2^l.rs$ (line 24). For $W_1(X)$, T_1 fails to acquire an exclusive lock on X, thus T_1 aborts (line 26, step ③). Upon commit of T_2, T_2 issues an $Atomic^D$ call to release the lock on X by reseting $X^m.lock$ from 1 to 0 (line 31, step ④). In this and the following examples, for illustration purposes, we show values of $X^m.lock$ in all steps, and changes are highlighted in red.

Wound-Wait [47] is another variant of 2PL algorithm. Similar to No-Wait, for any data item X, it always tries to acquire a lock on X before every read/write on X. Upon a conflict of a transaction T with another transaction $\overline{T}$, contrary to No-Wait that aborts T immediately, Wound-Wait does not abort T. Instead, it assigns transactions with priorities a priori. If $\overline{T}$ has lower priority, then $\overline{T}$ aborts (namely *Wound*); otherwise, T waits for the lock that $\overline{T}$ holds to release (namely *Wait*).

To re-implement Wound-Wait using RDMA, it also requires (1) acquiring remote locks, (2) noticing other transactions to abort, (3) doing remote reads/writes, and (4) releasing remote locks. In Wound-Wait, it is not possible to apply the same principles used in No-Wait for lock/unlock operations because an additional

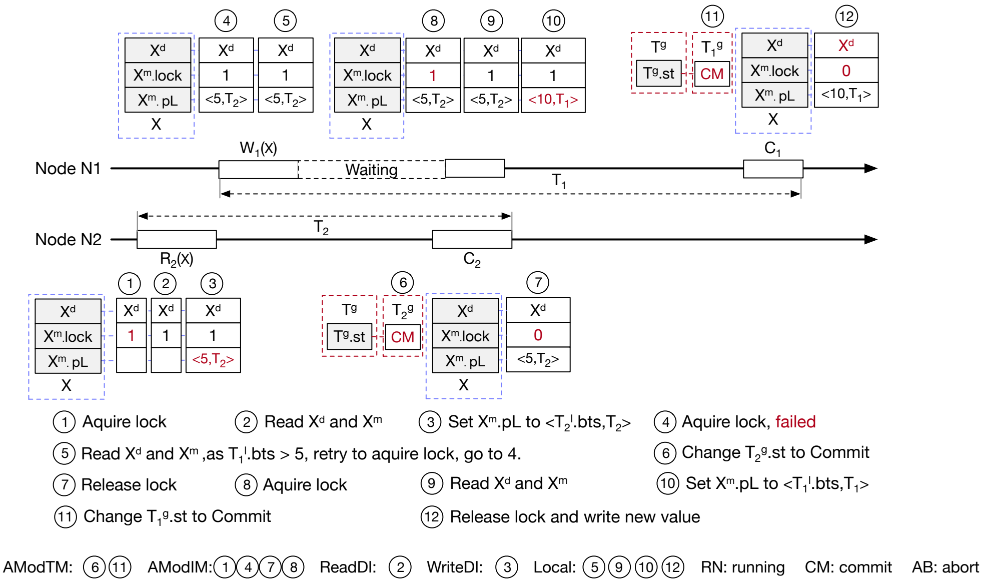


Fig. 4 An example of RDMA-Wound-Wait.

metadata $X^m.pL$ needs to be maintained to record the priorities of the transactions that own the lock on X. But for the logic (2) that notices other transactions to abort, it only modifies the state of remote transactions, satisfying the requirement of the One-Cas principle.

X^m includes two fields: (1) lock meta $X^m.lock$, and (2) the list $X^m.pL$, each item of which is a pair $\langle\ T^l.bts{,}T.Tid\ \rangle$ where T is a transaction with an exclusive or a shared lock on X. In our design, when a new transaction T is granted with a lock on X, a pair $\langle\ T^l.bts{,}T.Tid\ \rangle$ will be put into an empty slot of $X^m.pL$; when T commits, $\langle\ T^l.bts{,}T.Tid\ \rangle$ will be removed from $X^m.pL$ accordingly. In the real implementation, the size of $X^m.pL$ is set to be fixed. Besides, T^l in Wound-Wait includes the beginning timestamp ($T^l.bts$) of transaction T, and $T^l.rs$ and $T^l.ws$. T^g additionally maintains a transaction status $T^g.st$, which is *running* (RN), *committed* (CM) or *aborted* (AB). Key functions of the re-implementation called RDMA-Wound-Wait are shown in Algorithm 5.

Function *LockWithWound* is used to acquire remote locks, based on $Atomic^D$, $Read^D$ and $Write^D$ primitives. First, we issue *AcqIMExclock* (given in Algorithm 4) to acquire a remote lock on data item X with $X.PK = PK$ (line 2). If the lock is granted, we then read X using $Read^D$, set the lock owner on X to T, and perform a remote write of X using $Write^D$ to declare that a new lock on X is generated by T (line 7–9). Note, X's lock owner, maintained in $X^m.pl$, is a pair that consists of T's beginning timestamp $T^l.bts$ and ID $T.Tid$. If the lock is not granted, we perform a remote read of the lock owner $\overline{T}$ on X using $Read^D$ (lines 3–4), check the priority between T and $\overline{T}$ (line 5). If $\overline{T}$ has a lower priority (a larger transaction beginning timestamp means a lower priority), we let $\overline{T}$ abort asynchronously by issuing a single $Atomic^T$ to remote update AB on status $\overline{T}^g.st$ (line 6). Remind that if $\overline{T}^g.st$ is AB or CM, meaning that $\overline{T}$ aborts or commits, the remote write on status $\overline{T}^g.st$ would fail. In this case, because $\overline{T}$ already aborts or commits, locks owned by $\overline{T}$ would be released very soon. We keep retrying to acquire remote locks until the lock is granted to $\overline{T}$. *In the real implementation, when we issue a primitive call, we can switch to another co-routine within the same thread to reduce idle CPU time.*

Function *Read*, *Write* and *Commit* in RDMA-Wound-Wait are similar to those in RDMA-No-Wait except that *LockWithWound* is used instead of *AcqIMExcLock*. Besides, for *Commit*, if we cannot perform a remote update by changing the status $T^g.ST$ from RN to CM (e.g., T aborts), we abort T (line 12). In the real implementation, as an optimization to release locks as early as possible, we can periodically check $T^g.st$, and abort T if $T^g.st$ is AB. In the below example, we present how RDMA-Wound-Wait works.

Consider the example in Figure 3. Suppose $T_1^l.bts = 10$ and $T_2^l.bts = 5$. We show every step of executing RDMA-Wound-Wait in Figure 4. For $R_2(X)$, T_2 acquires a remote lock on X, does a remote read of X, sets the lock owner, and does a remote write of X (step ① – ③). For illustration purposes, we present the values of metadata including X^d, $X^m.lock$, and $X^m.pL$, as well as $T^g.st$ if necessary. Changes in these values are highlighted in red. For $W_1(X)$, T_1 fails to acquire the lock (step ④). Thus, T_1 does a remote read of the lock owner T_2, and checks the priority between T_1 and T_2 (step ⑤). Because T_1 has a lower priority, T_1 keeps retrying to acquire the lock until T_2 commits and releases the lock (step ⑥ – ⑦). T_1 acquires a remote lock on X, does a remote read of X, sets the lock owner, and does a remote write of X (step ⑧ – ⑩). Finally, T_1 commits and releases the lock (step ⑪ – ⑫).

Wait-Die [47] is another variant of 2PL algorithm. It follows the same logic as Wound-Wait, except for conflict handling. Upon a conflict on data item X of a transaction T with another transaction $\overline{T}$ that holds the lock on X, if T has a higher priority, then T waits for transaction $\overline{T}$ to release the lock on X (namely *Wait*); otherwise, T aborts (namely *Die*). Compared with RDMA-Wound-Wait, the re-implementation (called RDMA-Wait-Die) of Wait-Die can be achieved by replacing the comparison operator "<" by ">" (line 5), and Function call $Atomic^T$ by *Return Abort* (line 6) in algorithm 5.

Algorithm 6: RDMA-T/O

```
Function UpdateRTS($PK, X, T$):
    if $X^m.rts$ < $T^l.bts$ then
        $Atomic^D(PK, MT\_RTS, X^m.rts\ , T^l.bts\ )$;
        $X^m.rts \leftarrow T^l.bts$;
        if $X \neq Read^D(PK)$ then return $false$;;
    return $true$;
Function Read($PK, T$):
    $X \leftarrow Read^D(PK)$;
    if $X^m.wts > T^l.bts$ or $X^m.lock \neq 0$ then
        Abort $T$;
    if $\neg UpdateRTS(X, PK)$ then Abort $T$;
    $T^l.bL \leftarrow X^m.wL \cup T^l.bL$;
    $T^l.rs \leftarrow \{X\} \cup T^l.rs$; return $X$;
Function Write($PK, T, newV$):
    if $\neg$ $Atomic^D(PK, MT\_LATCH, 0, T.Tid)$ then
        Abort $T$;
    $X \leftarrow Read^D(PK)$;
    if max$\{X^m.rts, X^m.wts\ \} > T^l.bts$ then
        $Atomic^D(PK, MT\_LATCH, T.Tid, 0)$;
        Abort $T$;
    $T^l.ws \leftarrow \{X\} \cup T^l.ws$;
    $X^m.wts \leftarrow T^l.bts$; $X^d \leftarrow newV$;
    $T^l.bL \leftarrow X^m.wL \cup T^l.bL$;
    $X^m.wL \leftarrow X^m.wL \cup \{T.Tid\ \}$;
    $X^m.latch \leftarrow 0$; $Write^D(PK, X)$;
Function CascadingAbortCheck($T$):
    foreach $\overline{Tid} \in T^l.bL$ do
        do $\overline{T} \leftarrow Read^T(\overline{Tid})$ ;
        while $\overline{T}^g.st = RN$;
        if $\overline{T}^g.st = AB$ then
            $T^g.st \leftarrow AB$; return $false$;
    $T^g.st \leftarrow CM$; return $true$;
Function Commit($T$):
    if $\neg CascadingAbortCheck(T)$ then Abort $T$ ;
    foreach $X \in T^l.ws$ do
        while $\neg Atomic^D$(X.PK, $MT\_LATCH$, 0,$T.Tid$);
        $\overline{X} \leftarrow Read^D(X.PK)$;
        $\overline{X}^m.wL \leftarrow \overline{X}^m.wL$ - $T.Tid$;
        $\overline{X}^m.latch \leftarrow 0$; $Write^D(PK, \overline{X})$;
```

6.2.2 Timestamp-based Algorithms

T/O orders transactions based on their beginning timestamps ($T^l.bts$). If the execution order of the transactions does not match their beginning timestamp order, one of them needs to be aborted. To compare the timestamp, each data item maintains the maximum beginning timestamp of the transactions that have ever read/written X ($X^m.rts/X^m.wts$). For any read on data item X of transaction T, if there does not exist any conflicts, we update $X^m.rts$ to $\max\{X^m.rts, T^l.bts\}$; for any write on X of T, if there does not exist any conflicts, we update $X^m.wts$ to $\max\{X^m.wts, T^l.bts\}$. Upon a conflict of T with some other transaction $\overline{T}$ over X, we abort T if $T^l.bts < \overline{T}^l.bts$, meaning that T is supposed to be ordered before $\overline{T}$ but T reads/writes X that $\overline{T}$ has ever written;[2] otherwise, T must wait to commit/abort after $\overline{T}$ commits/aborts to guarantee correctness.

To boost T/O using RDMA, it is necessary to re-implement its logic that performs (1) remote reads/writes on data item X, (2) remote update of $X^m.rts$ or $X^m.wts$, and (3) wait-commit or cascading abort. In logic (2) of RDMA-T/O, when updating $X^m.rts$, only 8 bytes of metadata are modified, and the new value of $X^m.rts$ needs to be calculated against the old value, satisfying the requirement of the Read-Cas principle. Unfortunately, while a remote transaction is modifying $X^m.rts$, another write transaction may simultaneously modify other metadata $X^m.wts$. Using the Read-Cas principle alone cannot ensure the atomicity of this modification. To address this issue, we adopt the Double-Read principle to ensure atomicity. This principle avoids the need for an explicit latch and ensures the atomicity of the read operation. Furthermore, since logic (3) does not modify the remote MetaTxn metadata, we can use the One-Read principle to implement it.

The metadata X^m in T/O maintains four fields: (1) $X^m.latch$, the latch of X to prevent concurrent modification, (2) $X^m.rts$/(3) $X^m.wts$, the maximum beginning timestamp of the transactions that have ever read/written X, and (4) a list $X^m.wL$, each item of which is the ID of an uncommitted transaction that

[2] Specifically, for any read of T, we abort T if $T^l.bts < X^m.wts$, and for any write of T, we abort T if $T^l.bts < X^m.rts$ or $T^l.bts < X^m.wts$.

has ever written X. Besides $T^l.rs$, $T^l.ws$, T^l maintains the beginning timestamp $T^l.bts$ of T, and an extra list $T^l.bL$, recording transactions that are ordered before T. T^g additionally maintains a transaction status $T^g.st$, which is *running* (RN), *committed* (CM) or *aborted* (AB). Key functions of the re-implementation called RDMA-T/O are shown in Algorithm 6.

Function *Read* is used to do remote read using the Double-Read principle. We first read X by issuing $Read^D$ (line 8) and check whether X is readable by T (line 9). If X is not readable by T, we abort T (lines 10); otherwise, we try to do a remote update on $X^m.rts$ using $T^l.bts$, and perform another remote read on X to check whether the remote update is successful (lines 11, 2–6). If the update fails or the results of the two $Read^D$operations are different, we abort T; otherwise, we perform a local update on $T^l.bL$ (the dependent transactions of T) and store X in the read set $T^l.rs$ (lines 12–14).

Function *Write* is used to do remote write. We first use $Atomic^D$ to acquire latch, then issue $Read^D$ to read back X and check whether X is writable by comparing max$\{X^m.rts$,$X^m.wts$ $\}$ with $T^l.bts$ (lines 15–20). If max$\{X^m.rts$,$X^m.wts$ $\} > T^l.bts$, meaning X is not writable by T, then we release the latch and abort T (lines 18–20); otherwise, it means that X is writable by T. We then store the original X to the write set $T^l.ws$ for restoring X in case that T aborts (line 21). Subsequently, we do a local update on $X^m.wts$, X^d, $T^l.bL$, $X^m.wL$ (the running transactions with writes on X), $X^m.latch$, and finally we apply the local updates on remote X by issuing $Write^D$ (lines 22–25).

Upon committing transaction T, it is necessary to execute function *CascadingAbortCheck* to check the status of each dependent transaction, maintained in $T^l.bL$, of T (lines 27–31). If any of them aborts, we would make a cascading abort of T (lines 30–31); if all of them commit, we first set the status $T^g.st$ of T to CM (line 32), use a repeated invocation of $Atomic^D$ to acquire the latch of X (line 36), and do a remote update of $\overline{X}^m$ by removing $T.Tid$ from $\overline{X}^m.wL$ for each data item $\overline{X}$ that T has ever written (lines 37–39). Upon abort of transaction T, it is necessary to restore its modifications maintained in $T^l.ws$ on each data item X that has ever been written. Note, if a dependent transaction of T aborts, and restores X that T has ever written, in this case, T cannot restore X; otherwise, the value of X would be restored incorrectly. For example, suppose there exists a data item X with $X^d = 1$. Transaction T_1 first updates X^d to 2, and transaction T_2 then updates X^d to 3. Subsequently, T_1 aborts, and restore X^d to 1. Because the abort of T_1 causes a cascading abort of T_2, T_2 aborts. In this case, if T_2 does a restore of X which changes X^d to 2, then X would be set in an incorrect value.

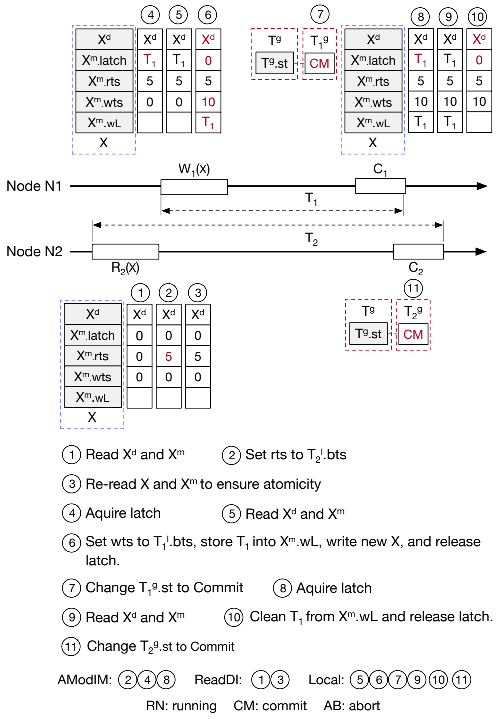


Fig. 5 An example of RDMA-T/O.

Discussion. One potential limitation of RDMA-T/O is that cascading aborts could waste CPU cycles. To eliminate cascading aborts, one possible solution is to postpone the writes of each transaction T until the commit of T. In this case, upon any read or write of X from T, if there exists another uncommitted transaction $\overline{T}$ with a smaller timestamp that writes X, T must wait until $\overline{T}$ commits. To notify the transactions that wait for $\overline{T}$, for each data item X, it is necessary to additionally maintain two lists, $X^m.prL$ and $X^m.pwL$ that record the pending transactions with reads and writes on X, respectively. After $\overline{T}$ commits, we sequentially check each X that $\overline{T}$ has written, and the transaction in $X^m.prL \cup X^m.pwL$ with the smallest timestamp is scheduled to execute, while the other transactions still need to wait. Although the above solution can eliminate cascading aborts, it instead incurs other potential overheads, e.g., the prohibitive maintenance overhead of $X^m.prL$ and $X^m.pwL$ for each data item X, as well as a large amount of CPU idle time. For comparison, we follow the work proposed by P.A.Bernstein and N.Goodman [8] to adjust RDMA-T/O without cascading abort, and report the experimental evaluation over the two implementations in Section 7.6. For illustration purposes, we present how RDMA-T/O works in the below example.

Consider the example in Figure 3. We show every step of executing RDMA-T/O in Figure 5. For $R_2(X)$, T_2 does a remote read on X, sets $X^m.rts$ to $T_2^l.bts$ by issuing $Atomic^D$, and does another remote read to ensure the atomicity (step ① – ③). For $W_1(X)$, T_1 acquires a remote latch on X, does a local read on X, sets $X^m.wts$ and $X^m.wL$, and does a local write of X (step ④ – ⑥). For C_1, T_1 sets $T^g.st$ to CM and removes T_1 from $X^m.wL$ (step ⑦ – ⑩). Finally, T_2 commits and set $T^g.st$ to CM (step ⑪).

In **MVCC**, each data item X has multiple versions that are denoted as $X_1, \ldots, X_k$, where X_k is the last version of X. Concurrent writes on X from two transactions are not allowed, but concurrent read/write or write/read on X from two transactions are allowed, where the read operation reads an older version and the write operation creates a new version so that read and write conflict can be avoided. Yet, simply applying MVCC to do concurrency control could lead to Write Skew [6] data anomaly. To achieve serializability, it is necessary to impose either T/O, OCC, or 2PL on MVCC so that it can produce a total order of concurrent transactions. In this paper, because MVCC mechanism is closely associated with timestamps, we enhance MVCC with T/O to achieve serializability. For ease of illustration, MVCC refers to MVCC plus T/O where the context is clear.

To guarantee serializability, for any write on X of transaction T without any conflicts, T creates a new version X_{k+1}, updates $X_{k+1}^m.wts$ to $\max\{X_{k+1}^m.wts, T^l.bts\}$, and uses meta $X_k^m.wid$ and $X_{k+1}^m.wid$ to prevent concurrent writes on X, which store the ID of uncommitted transaction that has written current version X_{k+1}; for any read on X of T without any conflicts, T reads the version X_i with the largest $X_i^m.wts$ smaller than $T^l.bts$, and updates $X_{k+1}^m.rts$ to $\max\{X_{k+1}^m.rts, T^l.bts\}$. Upon either a read/write or a write/read conflict of T with another transaction $\overline{T}$ on X_i, we abort T if $T^l.bts < \overline{T}^l.bts$ where $\overline{T}^l.bts$ is maintained as either $X_i^m.rts$ or $X_i^m.wts$. Upon a write-write conflict on X, we simply abort T.

To boost MVCC using RDMA, it also requires performing (1) remote reads/writes on data item X, and (2) remote updates of $X_i^m.rts$ or $X_i^m.wts$. For similar reasons as in RDMA-T/O, updating $X_i^m.rts$ in logic (2) can be optimized by applying the Double-Read principle. Besides, to reduce the number of primitives, we fixed the version number of each data item so that we could obtain the full versions of a data item by using a single primitive.

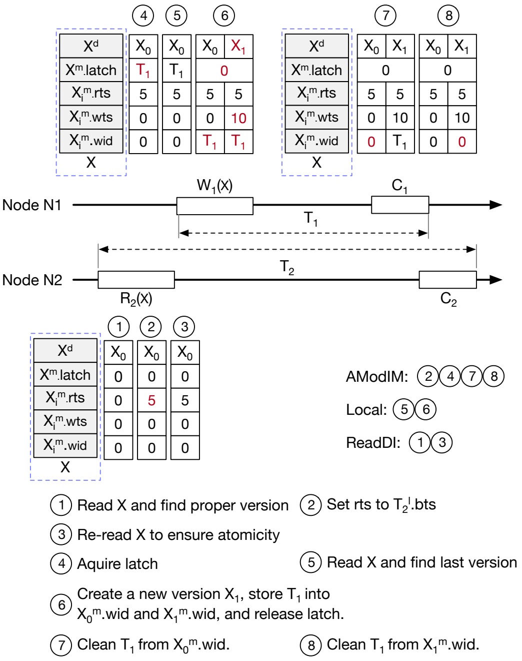
By doing this, a data item X keeps a fixed number (e.g. 4) of version slots, each of which stores a version X_i. In X, its metadata X^m maintains a latch meta $X^m.latch$ to prevent concurrent modification. X_i^m of each version X_i maintains three fields: (1) $X_i^m.rts$/(2) $X_i^m.wts$, the maximum beginning timestamp of the transactions that have ever read/written X_i, and (3) $X_i^m.wid$, the ID of the uncommitted transaction that has written current version X_n. T^l in MVCC includes read set $T^l.rs$, write set $T^l.ws$ of transaction T, and a beginning timestamp ($T^l.bts$) of transaction T. With these above metadata, we use Algorithm 7 to introduce our re-implementation called RDMA-MVCC.

Algorithm 7: RDMA-MVCC

```
1  Function UpdateVersionRTS($PK, X, T$):
2      if $X_i^m.rts < T^l.bts$ then
3          $Atomic^D(PK, MT\_X_i\_RTS, X_i^m.rts\ ,T^l.bts)$;
4          $X_i^m.rts \leftarrow T^l.bts$;
5          if $X \neq Read^D(PK)$ then $Abort\ T$;
6  Function Read($PK, T$):
7      $X \leftarrow Read^D(PK)$; // $X$ includes its all versions;
8      $X_i \leftarrow getProperVer(X, T)$;
9      if $X_i{=}NULL$ or $X_k^m.wid \neq 0$ or $X^m.latch \neq 0$ then $Abort\ T$;
10     $UpdateVersionRTS(PK, X, T)$;
11 Function Write($PK, T, newV$):
12     if $\neg\ Atomic^D(PK, MT\_LATCH, 0, T.Tid)$ then $Abort\ T$;
13     $X \leftarrow Read^D(PK)$;
14     $X_k \leftarrow getLatestVer(X)$;
15     if $\max\{X_k^m.rts, X_k^m.wts\} > T^l.bts$ or $X_k^m.wid \neq 0$ then
16         $Atomic^D(PK, MT\_LATCH, T.Tid, 0)$;
17         $Abort\ T$;
18     create a new version $X_{l+1}$ of $X$ by setting $X_{k+1}^d$ to $newV$ ;
19     $X_{k+1}^m.wts \leftarrow T^l.bts$; $X_{k+1}^m.wid \leftarrow T.Tid$;
20     $X_k^m.wid \leftarrow T.Tid$; $X^m.latch \leftarrow 0$;
21     $T^l.ws \leftarrow \{X\} \cup T^l.ws$;
22     $Write^D(PK, X)$;
23 Function Commit($X, T, newV$):
24     foreach $X \in T^l.ws$ do
25         $Atomic^D(X.PK,\ MT\_X_k\_WID,\ T.Tid,\ 0)$;
26         $Atomic^D(X.PK,\ MT\_X_{k+1}\_WID,\ T.Tid,\ 0)$;
```

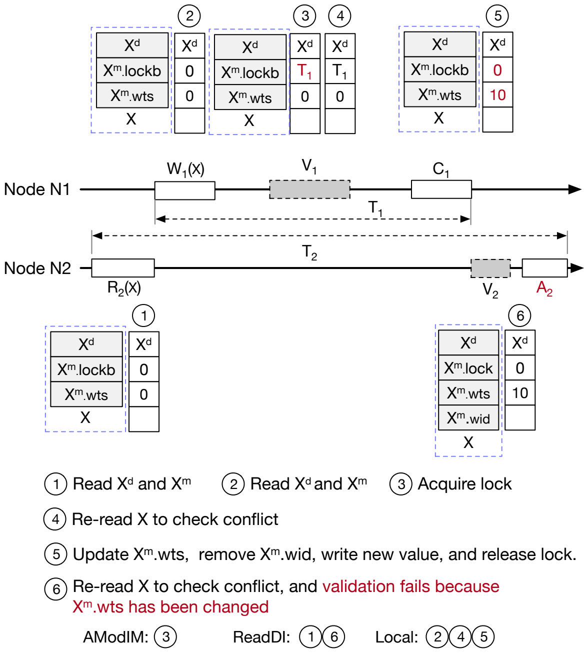
Function *Read* is used to do remote read based on primitives $Read^D$ and $Atomic^D$. We first perform a remote read to fetch X with all its versions by issuing $Read^D$. We follow MVCC to locally get the proper version X_i with the largest $X_i^m.wts$ smaller than $T^l.bts$ using Function $getProperVer$ (line 7–8). If X_i does not exist or the latch on X_i / X has been kept by another transaction, T aborts (line 9); otherwise, similar to T/O, we update the read timestamp of X_i using Function *UpdateVersionRTS* (line 1–5).

Function *Write* is used to write a new version of X based on primitives $Atomic^D$, $Read^D$ and $Write^D$.



Node N1
W1(X)
C1
T1
Node N2
T2
R2(X)
C2
X^d
X^m.latch
X_i^m.rts
X_i^m.wts
X_i^m.wid
X
AModIM: ②④⑦⑧
Local: ⑤⑥
ReadDI: ①③
① Read X and find proper version
② Set rts to T2^l.bts
③ Re-read X to ensure atomicity
④ Aquire latch
⑤ Read X and find last version
⑥ Create a new version X1, store T1 into X0^m.wid and X1^m.wid, and release latch.
⑦ Clean T1 from X0^m.wid.
⑧ Clean T1 from X1^m.wid.

Fig. 6 An example of RDMA-MVCC.



Node N1
W1(X)
V1
C1
T1
Node N2
T2
R2(X)
V2
A2
X^d
X^m.lockb
X^m.wts
X^m.lock
X^m.wid
X
① Read X^d and X^m
② Read X^d and X^m
③ Acquire lock
④ Re-read X to check conflict
⑤ Update X^m.wts, remove X^m.wid, write new value, and release lock.
⑥ Re-read X to check conflict, and validation fails because X^m.wts has been changed
AModIM: ③
ReadDI: ①⑥
Local: ②④⑤

Fig. 7 An example of RDMA-Silo.

Similar to RDMA-T/O, we first acquire a remote latch on X, perform a remote read on X with all its versions, and obtain its last version X_k using Function $getLatestVer$ locally (line 12–14). We then examine whether X is writable by T. If X is not writable, i.e., $\max\{X_k^m.rts, X_k^m.wts\} > T^l.bts$, or $X_k^m.wid \neq 0$, we abort T (line 15–17). If X is writable, we then locally create a new version X_{k+1}, update $X_{k+1}^m.wts$, exclusively lock versions X_k and X_{k+1} by setting $X_k^m.wid$ and $X_{k+1}^m.wid$ to $T.Tid$, copy X into $T^l.ws$, and we finally apply the local updates on remote X by issuing $Write^D$ (line 18–22). Note that if there is no available version slot in the data item X, we will choose to override the oldest version in X to create the new version X_{k+1} (line 18).

Upon commit of T, we only need to release locks ($X_k^m.wid$ and $X_{k+1}^m.wid$) on versions X_k and X_{k+1} and for each X in $T^l.ws$ by issuing $Atomic^D$ (line 24,25), in which $MT\_X_k\_WID$ presents the meta $X_k^m.wid$ and $MT\_X_{k+1}\_WID$ presents the meta $X_{k+1}^m.wid$. Upon abort of T, we additionally remove X_{k+1} located in the remote node for each data item X in $T^l.ws$. For illustration purposes, we present how RDMA-MVCC works in the below example.

Consider the example in Figure 3. We show every step of executing RDMA-MVCC in Figure 6. For $R_2(X)$, T_2 does a remote read on X, finds a proper version X_0, and sets $X_0^m.rts$ to $T_2^l.bts$ by issuing $Atomic^D$, and does another remote read to ensure the atomicity (step ① – ③). For illustration purposes, we present the values of metadata including X_i^d, $X_i^m.rts$, $X_i^m.wts$ and $X_i^m.wid$. Changes in these values are highlighted in red. For $W_1(X)$, T_1 acquires a remote latch first on X, does a local read on X, creates a new version X_1, sets $X_0^m.wid$, $X_1^m.wid$ and $X_1^m.wts$, and does a local write to apply these local update on X (step ④ – ⑥). Finally, T_1 commits, and cleans itself from $X_0^m.wid$ and $X_1^m.wid$ by using two $Atomic^D$ (step ⑦ – ⑧).

6.2.3 Optimistic Algorithms

Silo [59] is a classic optimistic concurrency control algorithm. In Silo, each transaction T is scheduled to execute through three phases: read/write phase, validation phase and commit/abort phase. In the read/write phase, for each read of X, we store X to read set $T^l.rs$; for each write of X, we store X to write set $T^l.ws$. In the validation phase, $\forall X \in T^l.ws$, it needs to acquire an exclusive lock on X and check whether X has been modified by other transactions. For $\forall X \in T^l.rs$, it checks whether X has been modified as well. If T cannot acquire all the locks successfully or if there exists a data item that has been modified by other transactions, we abort T and release all the locks held by T; otherwise, we commit T, and $\forall X \in T^l.ws$, we accordingly update X^d and $X^m.wts$.

To boost Silo using RDMA, we re-implement its logic that (1) do remote reads/writes, (2) acquire remote locks and validate data items in write set, and (3) validate data items in read set. For the logic of (1) and (3), they satisfy the requirement of the One-Read principles as they do not modify metadata. In logic (2), Silo uses only the exclusive lock mechanism, satisfying the requirement of One-Cas principle. To optimize the commit phase in the Silo algorithm, we can use the One-Write principle, as in the No-Wait algorithm.

To discuss the implementation of Silo in detail, we first design the metadata X^m in Silo, including two fields: (1) lock metadata $X^m.lockb$, recording the ID of a transaction that is currently granted with an exclusive lock on X; and (2) $X^m.wts$, the maximum commit timestamp of transactions that have ever written X. Besides $T^l.rs$ and $T^l.ws$, T^l includes the commit timestamp ($T^l.cts$) of transactions T. Then we discussed the key functions of the re-implementation called RDMA-Silo in Algorithm 8.

Functions *Read* and *Write* are used to perform the remote read/write on data item X by using One-Read principle. For *Read*, we read remote X by using a $Read^D$, and add it to the local read set $T^l.rs$ (line 2). For *Write*, we read remote X by using a $Read^D$, update X^d locally, and add X to local write set $T^l.ws$ (lines 4–5).

Function *Validation* is used to perform the operation in the validation phase. First, we obtain a commit timestamp of T locally. Then, $\forall X \in T^l.ws$, we try to acquire an exclusive lock on X following the One-Cas principle (line 9). If the lock acquisition fails, T aborts; otherwise, we continue to validate T. Subsequently, $\forall X \in T^l.ws \cup T^l.rs$, we examine whether X has been modified by other transactions based on a One-Read principle to read X again by using $Read^D$ (line 10–11). If X has been locked or modified, we abort T (lines 12–13).

Upon commit of T, $\forall X \in T^l.ws$, we make a single remote write by issuing $Write^D$ to release the lock on X, as well as update X^d. Upon abort of T, we only need to release all the locks on $\forall X \in T^l.ws$. For illustration purposes, we then present how RDMA-Silo works in the below example.

Consider the example in Figure 3. We present values of metadata including X^d, $X^m.lockb$, and $X^m.wts$ at every step of executing RDMA-Silo in Figure 7. For $R_2(X)$, T_2 does a remote read on X, and stores X in $T^l_2.rs$ (step ①). For $W_1(X)$, T_1 does a local read on X, updates X^d locally, and stores X in $T^l_1.ws$ (step ②). In the validation phase of T_1, first, T_1 acquires the lock on X by issuing a $Atomic^D$. Then, T_1 does a local read on X to ensure that X has not been modified by other transactions (step ③ − ④). Next, T_1 commits. T_1 updates X^d, $X^m.wts$, and releases the lock (step ⑤). In the validation phase of T_2, T_2 does a remote read on X, and finds that X has been modified by T_1. Thus, we abort T_2 (step ⑥).

Algorithm 8: RDMA-Silo

```
1  Function Read($PK, T$):
2      $X \leftarrow Read^D$(PK); $T^l.rs \leftarrow \{X\} \cup T^l.rs$;
3  Function Write($PK, T, newV$):
4      $X \leftarrow Read^D$(PK); $X^d \leftarrow newV$;
5      $T^l.ws \leftarrow \{X\} \cup T^l.ws$;
6  Function Validation($T$):
7      $T^l.cts \leftarrow$ get the current timestamp;
8      foreach $X \in T^l.ws$ do
9          if $\neg Atomic^D(PK, MT\_LOCKB, 0, T.Tid)$ then $Abort\ T$;
10     foreach $X \in T^l.rs \cup T^l.ws$ do
11         $\overline{X} \leftarrow Read^D(X.PK)$;
12         if $\overline{X}^m.lockb \neq 0$ and $\overline{X}^m.lockb \neq T.Tid$ then $Abort\ T$;
13         if $\overline{X}^m.wts \neq X^m.wts$ then $Abort\ T$;
14 Function Commit($T$):
15     foreach $X \in T^l.ws$ do
16         $X^m.wts \leftarrow T^l.cts$; $X^m.lockb \leftarrow 0$;
17         $Write^D$(X.PK, $X$);
```

Algorithm 9: RDMA-MaaT (Part 1)

```
Function Read($PK, T$):
    do $res \leftarrow Atomic^D(PK, MT\_LATCH, 0, T.Tid)$ ;
    while $\neg res$;
    $X \leftarrow Read^D(PK)$;
    $X^m.rL \leftarrow \{T.Tid\} \cup X^m.rL$ ;
    $X^m.latch \leftarrow 0$; $Write^D(PK, X)$;
    $T^l.rs \leftarrow \{X\} \cup T^l.rs$;
    $T^l.aL \leftarrow X^m.wL \cup T^l.aL$;
    $T^l.gwts \leftarrow max(T^l.gwts,\ X^m.wts + 1)$;
Function Write($PK, T, newV$):
    do $res \leftarrow Atomic^D(PK, MT\_LATCH, 0, T.Tid)$ ;
    while $\neg res$;
    $X \leftarrow Read^D(PK)$;
    $X^m.wL \leftarrow \{T.Tid\} \cup X^m.wL$ ;
    $X^m.latch \leftarrow 0$; $Write^D(PK, X)$;
    $X^d \leftarrow newV$; $T^l.ws \leftarrow \{X\} \cup T^l.ws$;
    $T^l.bL \leftarrow \{X^m.rL\} \cup T^l.bL$;
    $T^l.oL \leftarrow \{X^m.wL\} \cup T^l.oL$;
    $T^l.grts \leftarrow max(T^l.grts,\ X^m.rts + 1)$;
    $T^l.gwts \leftarrow max(T^l.gwts,\ X^m.wts + 1)$;
Function AcqTMExcLock($Tid, T$):
    $r \leftarrow Atomic^D(Tid, MT\_LATCH, 0, T.Tid)$;
    return $r$;
Function RlsTMExcLock($Tid, T$):
    $Atomic^D(Tid, MT\_LATCH, T.Tid, 0)$);
```

MaaT [30] is a recently proposed optimistic concurrency control algorithm. It introduces a timestamp interval $[T^g.lb, T^g.ub]$ for each transaction T, and follows the idea of dynamic timestamp adjustment of $[T^g.lb, T^g.ub]$ [11] to order transactions. If T reads a data item X written by another transaction $\overline{T}$, then we must guarantee the order $\overline{T} \rightarrow T$ (note, only if a transaction $\overline{T}$ commits, its writes can be read by other transactions).

For any two concurrent transactions T and $\overline{T}$ that read and write the same X, respectively, if T does not read X written by $\overline{T}$, we must guarantee the order $T \rightarrow \overline{T}$. Similarly, for two concurrent transactions T and $\overline{T}$ that write the same X, if T enters validation phase first, we must guarantee the order $T \rightarrow \overline{T}$; otherwise, we must guarantee the order $\overline{T} \rightarrow T$. To preserve order $\overline{T} \rightarrow T$, in the validation phase, MaaT always guarantees that the timestamp intervals of T and $\overline{T}$ are disjoint by adjusting $\overline{T}^g.ub < T^g.lb$. Upon commit of T, if T has a legal timestamp interval (i.e. $T^g.lb \leq T^g.ub$), T commits; otherwise, T aborts.

To boost MaaT using RDMA, we re-implement its logic that (1) perform remote reads/writes on data item X, and (2) remotely adjust timestamp intervals of transactions. Because each operation in MaaT needs to modify more than one metadata, we cannot apply any principle in the re-implementation called RDMA-MaaT.

X^m in MaaT includes five fields: (1) latch meta $X^m.latch$, (2) $X^m.rts$/(3) $X^m.wts$, the maximum commit timestamp of the transactions that have ever read/ written X, (4) a list $X^m.rL$, each item of which stores the ID of a running transaction that has ever read X, and (5) another list $X^m.wL$, each item of which records the ID of a running transaction that has ever written X. To satisfy the requirement that obtains X^m by only a one-sided verb, we set both $X^m.rL$ and $X^m.wL$ to be of fixed length. Besides $T^l.rs$, $T^l.ws$ and $T^l.cts$, T^l includes five fields additionally: (1) a list $T^l.bL$, each item of which records a transaction that needs to be ordered before T, (2) the second list $T^l.aL$, each item of which records a transaction that needs to be ordered after T, (3) the third list $T^l.oL$, each item of which stores a transaction that is not in $T^l.bL$ and $T^l.aL$, but has modified the same data items with T, (4) $T^l.grts$, the maximum $X^m.rts$ of each X that has been written by T, and (5) $T^l.gwts$, the maximum $X^m.wts$ for each X that has been read or written by T. T^g includes a latch meta $T^g.latch$, the status of transaction $T^g.st$ and a timestamp interval $[T^g.lb$,$T^g.ub\,]$, which is dynamically adjusted in the validation phase according to the order between T and its concurrent transactions. Key functions of the RDMA-MaaT are shown in Algorithm 9 and 10.

Function *Read* is used to do remote read based on $Atomic^D$, $Read^D$ and $Write^D$. For read, we first acquire a remote latch on X to prevent concurrent modifications (lines 2–3). To let other transactions know that T has ever read X, we issue $Read^D$ to do a remote read on X, locally update $X^m.rL$ (record running transactions that have ever read X) by $T.Tid$, and make a remote write of updating X^m as well as releasing the latch (lines 4–6). Subsequently, we perform a sequence of local operations that update read set $T^l.rs$ by X, $T^l.aL$ by $X^m.wL$, and $T^l.gwts$ by max$\{T^l.gwts$, $X^m.wts + 1\}$ (line 7–9). Remind, for two concurrent transactions that read and write the same data item, we must determine the order of them. For concurrent transactions that write X which is not read by T, we maintain these transactions in $T^l.aL$ that are ordered after T (line 8). In MaaT, only if a transaction commits, its writes can be read by other transactions. Thus, to preserve the order $\overline{T} \rightarrow T$ that T reads a data item X written by $\overline{T}$, we guarantee that $T^g.lb > \overline{T}^g.ub$ where $\overline{T}$ is a committed transaction. To do this, we maintain meta $X^m.wts$ to record the commit timestamp of the latest transaction that has ever written X. Besides, we do not explicitly store all transactions with their writes of data items read by T. Instead, we only maintain the greatest $X^m.wts$ (i.e., $T^l.gwts$) of each X^m that T reads to make sure $T^g.lb > T^l.gwts$ (line 9).

Function *Write* is used to do remote write based on $Atomic^D$, $Read^D$ and $Write^D$. We first acquire a remote latch on X to prevent concurrent modifications (lines 11–12). To let other transactions know that T attempts to write X, we issue $Read^D$ to do a remote read on X, locally update $X^m.wL$ (record running transactions that have ever written X) by $T.Tid$, and make a remote write of updating X^m as well as releasing the latch (lines 13–15). Subsequently, we perform a sequence of local operations that update write set $T^l.ws$ by X, $T^l.bL$ by $X^m.rL$, $T^l.oL$ by $X^m.wL$, $T^l.grts$ by max$\{T^l.grts, X^m.rts{+}1\}$, and $T^l.gwts$ by max$\{T^l.gwts$, $X^m.wts + 1\}$ (line 16–20). For concurrent transactions that have ever read X which is not written by T, we maintain these transactions in $T^l.bL$ that are ordered before T (line 17). For concurrent transactions that have ever written X, we maintain these transactions in $T^l.oL$ and determine the orders in the validation phase (line 18). For any committed transaction $\overline{T}$ that has read or written X, we must guarantee that $T^g.lb > \overline{T}^g.ub$ to preserve the order $\overline{T} \rightarrow T$. To do this, we additionally maintain $X^m.rts$ and $X^m.wts$ to record the commit timestamp of the latest transaction that has ever read or written X. Due to the similar reason given in Function *Read*, we maintain the maximum $X^m.rts$ and maximum $X^m.wts$ of X^m, which T writes, in $T^l.grts$ and $T^l.gwts$, respecitvely, to compute $T^g.lb$ and make sure $T^g.lb > \max\{T^l.grts, T^l.gwts\}$(line 19–20).

Upon validation phase of T, we use Function *Validation* to adjust the timestamp intervals to preserve transaction orders based on $Atomic^T$, $Read^T$ and $Write^T$. First, we update $T^g.lb$ to max$\{T^l.grts$, $T^l.gwts\}$ on the premise of obtaining a remote latch (lines 2–5). Then, we sequentially check every transaction $\overline{T}$ in $T^l.bL \cup$

$T^l.aL \cup T^l.oL$ (line 6–15). If $\overline{T}$ already commits, we adjust the timestamp interval of T locally based on Equation 1;

$$\begin{cases} T^g.lb = max(T^g.lb, \overline{T}^g.ub + 1) \text{ //if } \overline{T} \rightarrow T \\ T^g.ub = min(T^g.ub, \overline{T}^g.lb - 1) \text{ //if } T \rightarrow \overline{T} \end{cases} \tag{1}$$

otherwise, we adjust the timestamp interval of $\overline{T}$ remotely based on Equation 2.

$$\begin{cases} \overline{T}^g.lb = max(\overline{T}^g.lb, T^g.ub + 1) \text{ //if } T \rightarrow \overline{T} \\ \overline{T}^g.ub = min(\overline{T}^g.ub, T^g.lb - 1) \text{ //if } \overline{T} \rightarrow T \end{cases} \tag{2}$$

In Function *Commit*, first, we check whether T has a legal timestamp interval. If T has a legal timestamp interval, we set the status $T^g.st$ of T to CM, use $T^g.lb$ as the commit timestamp, and set $T^g.ub$ to $T^g.lb$ (line 17–21); otherwise, we abort T (line 22). Note that the timestamp interval of T could be adjusted by other transactions in their validation phases. To avoid this adjustment, we acquire an exclusive lock on T and set the status $T^g.st$ of T to CM. To let other transactions have the knowledge that T writes data items X and commits, we update X^d, $X^m.rts$, $X^m.wts$, and remove T from $X^m.wL$ by issuing $Atomic^D$, $Read^D$ and $Write^D$ (line 23–29). To let other transactions know that T reads data items X and commits, we update $X^m.rts$ and remove T from $X^m.rL$ by issuing the same three primitives (lines 30–35). Upon abort of T, we only need to set $T^g.st$ to AB and remove T from $X^m.rL$ and $X^m.wL$ using the same primitives in Function *Commit*. For illustration purposes, we present how RDMA-MaaT works in the below example.

Algorithm 10: RDMA-MaaT (Part 2)

```
1  Function Validation(PK, T):
2      do $res \leftarrow$ AcqTMExcLatch($T.Tid$);
3      while $\neg res$;
4      $T^g.lb \leftarrow max(T^g.lb,\ T^l.gwts,\ T^l.grts\ )$;
5      $T^g.latch \leftarrow 0$;
6      foreach $\overline{Tid} \in T^l.bL \cup T^l.aL \cup T^l.oL$ do
7          do $res \leftarrow$ AcqTMExcLatch($T.Tid$);
8          while $\neg res$;
9          do $res \leftarrow$ AcqTMExcLatch($\overline{Tid}$);
10         while $\neg res$;
11         $\overline{T} \leftarrow Read^T(\overline{Tid})$;
12         AdjustTsInterval($\overline{T}, T$);
13         // Equation 1 and 2;
14         $\overline{Tid}^g.latch \leftarrow 0$;
15         $Write^T(\overline{Tid},\overline{T})$; $T^g.latch \leftarrow 0$;
16 Function Commit(T):
17     do $res \leftarrow$ AcqTMExcLatch($T.Tid$);
18     while $\neg res$;
19     if $T^g.lb \leq T^g.ub$ then
20         $T^g.st \leftarrow CM$; $T^g.ub \leftarrow T^g.lb$;
21     $T^g.latch \leftarrow 0$;
22     if $T^g.st \neq CM$ then $Abort\ T$;
23     foreach $X \in T^l.ws$ do
24         while $\neg Atomic^D$(X.PK, $MT\_LATCH$,
             0,$T.Tid$);
25         $\overline{X} \leftarrow Read^D$(PK); $\overline{X}^d \leftarrow X^d$;
26         $\overline{X}^m.wts \leftarrow max(\overline{X}^m.wts\ ,T^g.lb\ )$;
27         $\overline{X}^m.rts \leftarrow max(\overline{X}^m.rts\ ,T^g.lb\ )$;
28         $\overline{X}^m.wL \leftarrow \overline{X}^m.wL$ - $T.Tid$;
29         $\overline{X}^m.latch \leftarrow 0$; $Write^D(PK, \overline{X})$;
30     foreach $X \in T^l.rs$ do
31         while $\neg Atomic^D$(X.PK, $MT\_LATCH$,
             0,$T.Tid$);
32         $\overline{X} \leftarrow Read^D$(PK);
33         $\overline{X}^m.rts \leftarrow max(\overline{X}^m.rts\ ,T^g.lb\ )$;
34         $\overline{X}^m.rL \leftarrow \overline{X}^m.rL$ - $T.Tid$;
35         $\overline{X}^m.latch \leftarrow 0$; $Write^D(PK, \overline{X})$;
```

Consider the example in Figure 3. We present values of metadata including X^d, $X^m.latch$, $X^m.rts$, $X^m.wts$, $X^m.rL$ and $X^m.wL$ at every step of executing RDMA-MaaT in Figure 8. For $R_2(X)$, T_2 acquires a remote latch on X, does a remote read on X, and does a remote write to update $X^m.rL$ (step ① – ③). Then, T_2 locally stores X into $T^l.rs$ and sets $T_2^l.gwts$ to 1 ($X^m.wts$ +1). For $W_1(X)$, T_1 acquires a latch on X, does a local read on X, and does a local write to update $X^m.wL$ (step ④ – ⑥). Subsequently, T_1 locally stores X into $T^l.ws$, updates $T^l.bL$ to T_2, and sets $T_1^l.grts = T_1^l.gwts = 1$. When T_1 enters the validation phase, T_1 updates $T_1^g.lb$ to 1 according to $T_1^l.grts$ and $T_1^l.gwts$ (step ⑦ – ⑨), and updates $T_2^g.ub$ to 0 as T_2 needs to be ordered before T_1 (step ⑩ – ⑫). When T_2 enters the validate phase, T_2 updates $T_2^g.lb$ to 1 using the same three steps (step ⑬ – ⑮), and finds that $T_2^g.lb > T_2^g.ub$, so T_2 aborts. When T_1 enters the commit phase, because T_1 has a legal timestamp interval, first, T_1 sets $T_1^g.st$ to CM (step ⑯ – ⑰). Then, T_1 updates $X^m.rts$ and $X^m.wts$, removes T_1 from $X^m.wL$ and updates X^d (step ⑱ – ⑳). When T_2 enters the abort phase, T_2 sets $T_2^g.st$ to AB locally, and removes T_2 from $X^m.rL$ remotely (step ㉑ – ㉕). Due to the space limitation, we omit the step of releasing the latches, which can be combined in the remote write of X^m (the meta $x^m.latch$ highlighted in red).

Cicada [37] is a hybrid concurrency control algorithm that imposes OCC on MVCC to achieve serializability. Like Silo, in Cicada, each transaction T is scheduled to execute through three phases: read/write phase, validation phase and commit/abort phase. In the read/write phase, for any read on data item X of transaction T, contrary to MVCC returning the version X_i of X that is written by a committed transaction with the

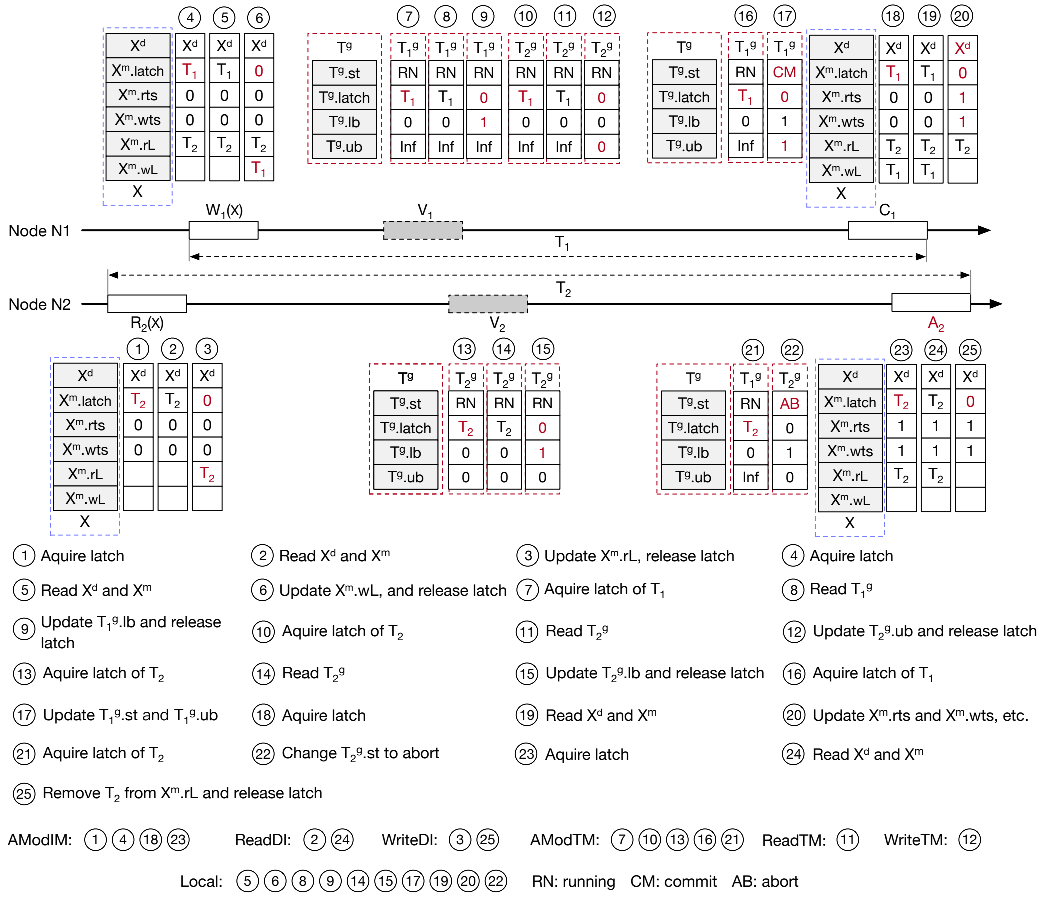


Fig. 8 An example of RDMA-MaaT.

largest write timestamp $X_i^m.wts$ smaller than the beginning timestamp $T^l.bts$ of T, Cicada simply returns the version X_i with the largest $X_i^m.wts$ smaller than $T^l.bts$. If X_i is written by an aborted transaction (i.e., the status of version X_i $X_i^m.st = AB$), it skips this invalid version and continues to find another version; if X_i is written by a running transaction (i.e., $X_i^m.st = RN$), T waits until the transaction that writes X_i commits or aborts. If the transaction aborts, it skips this invalid version and continues to find another version. After obtaining a proper version of X, it adds X to read set $T^l.rs$ of T. For any write on X, it obtains the last version X_k and aborts T if the transaction that writes X_k is still running; otherwise, it adds X to write set $T^l.ws$ of T. In the validation phase, $\forall X \in T^l.ws$, if there exists a newer version X_{k+1} written by some other transaction, T aborts; otherwise, it creates a new temporary version X_{k+1} for T; $\forall X \in T^l.rs$, if there exists a newer version written by some other transaction and can be read by T, T aborts; otherwise, it updates $X_i^m.rts$ by $\max\{T^l.bts, X_i^m.rts\}$. Upon commit of T, it updates the status of each version written by T to CM to make each version visible to other transactions.

To boost Cicada using RDMA, we need to re-implement the logic that (1) perform remote reads/writes, (2) create new versions of data items in the write set, and (3) do the remote validation of data items in the read set. For similar reasons as in RDMA-Silo, logic (1) can be optimized by the One-Read principle because it does not modify metadata. For logic (2) and (3), we use the basic approach to implement them because their logic is complex and requires modification of multiple metadata.

The metadata in Cicada is similar to that in MVCC, except that $X_n^m.st$ is used instead of $X_n^m.wid$ for each version X_n, where $X_n^m.st$ represents the status of the transaction that has written the current version X_n. Key functions of the re-implementation called RDMA-Cicada are shown in Algorithm 11.

Function *Read* is used to perform remote read on X based on primitive $Read^D$. We first perform a remote read of X with all its versions by issuing $Read^D$. We then obtain X_i locally with the largest $X_i^m.wts$ smaller than $T^l.bts$ using Function $getProperVer$ (line 3–4). If X_i does not exist, we abort T (line 5). If $X_i^m.st$ equals

Algorithm 11: RDMA-Cicada

```
1  Function Read($PK,T$):
2      do
3          $X \leftarrow Read^D(PK)$;
4          $X_i \leftarrow getProperVer(X,T)$;
5          if $X_i = NULL$ then $Abort\ T$;
6      while $X_i^m.st = RN$;
7      $T^l.rs \leftarrow \{X\} \cup T^l.rs$;
8  Function Write($PK,T,newV$):
9      $X \leftarrow Read^D(PK)$; $X_k \leftarrow getlatestVer(X)$;
10     if $X_k^m.wts > T^l.bts$ or $X_k^m.rts > T^l.bts$ or $X_k^m.st$
         $= RN$ then $Abort\ T$ ;
11     $T^l.ws \leftarrow \{\langle X,newV\rangle\} \cup T^l.ws$;
12 Function Validation($T$):
13     foreach $\langle X,newV\rangle \in T^l.ws$ do
14         if $\neg Atomic^D(X.PK, MT\_LATCH,\ 0, T.Tid)$
             then
15             $Abort\ T$;
16         $\overline{X} \leftarrow Read^D(X.PK)$;
17         $\overline{X_k} \leftarrow getlatestVer(\overline{X})$;
18         if $\overline{X_k} \neq X_k$ then
19             RlsIMExcLock(PK); $Abort\ T$;
20         create a new version $\overline{X}_{k+1}$ of $\overline{X}$ by setting
             $\overline{X}_{k+1}^d$ to newV;
21         $\overline{X}_{k+1}^m.st \leftarrow RN$;
22         $\overline{X}_{k+1}^m.wts \leftarrow T^l.bts$;
23         $\overline{X}^m.latch \leftarrow 0$; $Write^D(PK,\overline{X})$;
24     foreach $X \in T^l.rs$ - $T^l.ws$ do
25         if $\neg Atomic^D(X.PK, MT\_LATCH,\ 0, T.Tid)$
             then
26             $Abort\ T$;
27         $\overline{X} \leftarrow Read^D(X.PK)$;
28         $\overline{X}_i \leftarrow getProperVer(\overline{X},T)$;
29         if $\overline{X}_i \neq X_i$ then
30             RlsIMExcLock(PK);$Abort\ T$;
31         $\overline{X}_i^m.rts \leftarrow max(\overline{X}_i^m.rts\ , T^l.bts\ )$;
32         $\overline{X}^m.latch \leftarrow 0$; $Write^D(PK,\overline{X})$;
33 Function Commit($T$):
34     foreach $\langle X,newV\rangle \in T^l.ws$ do
35         $Atomic^D(PK, MT\_X_{k+1}\_ST,\ RN,\ CM)$;
```

to RN, we let T wait until $X_i^m.st$ is set to CM or AB (line 2–6); otherwise, we add X to $T^l.rs$.

In Function *Write*, we first perform a remote read on X with all its versions, and locally obtain its last version X_k using $getLatestVer$ (line 9). We then examine whether X is writable by T. If X is not writable, i.e., $\max\{X_k^m.rts, X_k^m.wts\} > T^l.bts$, or $X^m.st = RN$, we abort T (line 10); otherwise, we add X to $T^l.ws$ (line 11).

In the validation phase of T, $\forall X \in T^l.ws$, we first attempt to create a new version X_{k+1} of X. To do this, we acquire a latch on X to prevent concurrent modification (lines 14–15) and re-read X_k to identify the last version $\overline{X}_k$ (lines 16–17). If $\overline{X}_k$ does not match X_k, we abort T (line 18–19); otherwise, we create a new version X_{k+1}, update $X_{k+1}^m.st$ and $X_{k+1}^m.wts$, and we finally issue $Write^D$ to apply the local update on remote X as well as release the latch (line 20–23). We then do a remote validation of data items maintained in $T^l.rs$ only. To do this, we acquire a latch on X to prevent concurrent modification (lines 25–26) and re-read X to identify the proper version $\overline{X}_i$ (lines 27–28). If $\overline{X}_i$ does not match X_i that T reads before, we abort T; otherwise, we update $X_i^m.rts$ and release the latch by issuing $Write^D$ (lines 29–32).

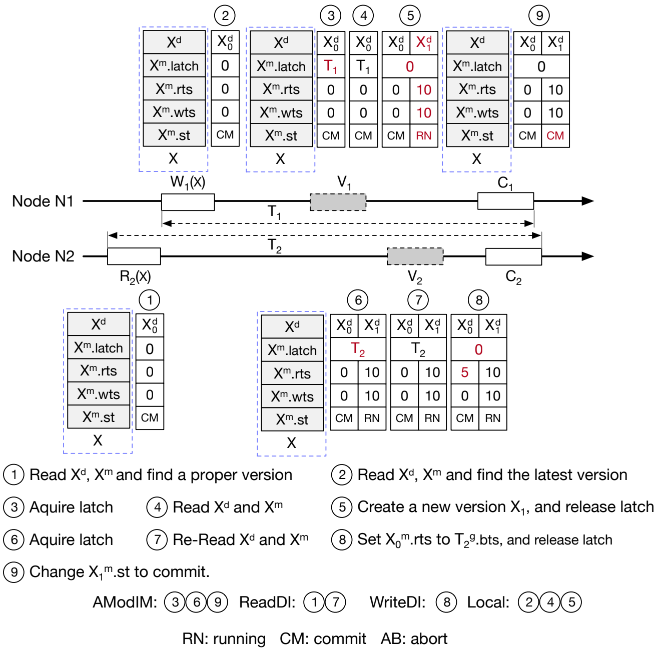


Fig. 9 An example of RDMA-Cicada.

Upon commit of T, for each data item X in $T^l.ws$, we set the status $X_{k+1}^m.st$ of X to CM (line 35), where $MT\_X_{k+1}\_ST$ denotes the meta $X_{k+1}^m.st$. Upon abort of T, we set the status $X_{k+1}^m.st$ of X to AB. For illustration purposes, we present how RDMA-Cicada works in the below example.

Consider the example in Figure 3. We present values of metadata including X_i^d, $X_i^m.latch$, $X_i^m.rts$, $X_i^m.wts$ and $X_i^m.st$ at every step of executing RDMA-Cicada in Figure 9. For $R_2(X)$, T_2 does a remote read on X, and obtains the version X_0 (step ①). For $W_1(X)$, T_1 does a remote read on X, and obtains the last version X_0 (step ②). In the validation phase of T_1, T_1 acquires the latch on X, re-reads X, creates a new version X_1 of X, and updates $X_1^m.rts$, $X_1^m.wts$ and $X_1^m.st$ (step ③ – ⑤). In the validation phase of T_2, T_2 acquires the latch on X, re-reads X, sets $X_0^m.rts$, and does a remote update on X (step ⑥ – ⑧). Finally, T_1 commits, and T_1 sets $X_1^m.st$ to CM using $Atomic^D$ (step ⑨).

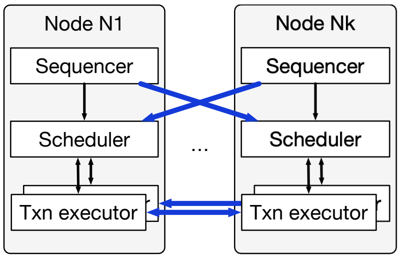


Fig. 10 Calvin overview.

6.2.4 Deterministic Algorithms

Calvin accepts and executes transactions in batches. In each batch, it determines the order of transactions in a first-come-first-serve manner. For each transaction T, if there does not exist any transaction that conflicts with T and is ordered before T, T is scheduled to execute. Note, in Calvin, no transaction is aborted because of conflict. To be specific, Calvin is designed with three components to do concurrency control.

- **Sequencer** collects the transactions in batches. For each batch, it determines the order of transactions, breaks every transaction into several sub-transactions, and distributes sub-transactions to schedulers based on the data items they access.
- **Scheduler** schedules sub-transactions to execute in a pre-defined order. All sub-transactions attempt to acquire locks on data items to be read/written. When different sub-transactions apply for locks on the same data item, the scheduler grants the locks to sub-transactions in a predetermined order. A sub-transaction is scheduled to execute if it acquires all locks.
- **Transaction executor** is used to execute sub-transactions. For each sub-transaction Ts, it performs local reads. For another sub-transaction that belongs to the same transaction with Ts, its writes may rely on reads of Ts, the transaction executor then synchronizes the reads to the other sub-transaction if necessary. Finally, the executor performs local writes and releases the acquired locks and commits.

There are two opportunities to optimize Calvin using RDMA: 1) distributing sub-transactions from sequencers to schedulers, and 2) synchronizing reads among executors. For illustration purposes, we show these two opportunities by the blue lines in Figure 10. For opportunity one, we implement a circular buffer following FaRM [21] to replace the original TCP/IP network message transmission mechanism. And for opportunity two, we design a fixed-length read set buffer stored in Ts^g for each sub-transaction Ts. To do this, when a sub-transaction Ts_1 on Node $N1$ synchronizes its read set to another sub-transaction Ts_2 on the remote node $N2$, we remotely write local read set into $Ts_2^g.N2.rs$ by issuing a $Write^D$. After Ts_2 gathers all read sets of remote nodes, Ts_2 can continue to perform execution.

Discussion. However, as reported in [31], network overhead is comparatively trivial and not the bottleneck of Calvin. Therefore, the above two optimizations cannot help effectively improve the system performance, we argue using RDMA to optimize Calvin cannot bring obvious benefits and verify this observation in the experiment section. The same reason in Calvin is also applicable for other deterministic algorithms such as Q-Store [43], LADS [65] and QueCC [44] whose network usage is quite limited.

6.3 Optimizations

We use four optimizations for concurrency control algorithms.

- **Coroutine.** One thread can have multiple coroutines, each of which executes transactions sequentially. Coroutines of the same thread are switched in a round-robin manner upon one coroutine is blocked by waiting for the results of RDMA verbs: Upon sending an RDMA verb from one coroutine, another coroutine of the same thread is switched to execute transactions. By using coroutines in Boost C++ library with low context switch overhead (about 20 ns), we can reduce the CPU idle time and hence improve $\mathcal{U}^e_{CPU}$.
- **Doorbell Batching (a.k.a. DB).** Usually, a verb takes one Memory Mapping I/Os (MMIOs). Instead, DB encapsulates multiple verbs into a batch and calls a single MMIO to send the beginning address of the batch. This address serves as a ringing doorbell to notify RNIC to fetch the batch through one or more DMAs. In this way, expensive MMIOs are replaced by a low-cost CPU and bandwidth-efficient DMA, therefore improving $\mathcal{U}^e_{CPU}$. Given a batch of verbs, DB works only if there is no dependency among these verbs. For example, we cannot batch $Read^D$ with $Atomic^D$ in Function *AcqIMShLock*(line 2–9 in Algorithm 4) because the inputs of $Atomic^D$ primitive rely on the result of $Read^D$ primitive. For this reason, we sequentially check the primitives evoked by the concurrency algorithms and batch continuous primitives without dependency using DB. For example, we batch continuous *AModIM* and *ReadDI* at line 12–13 in algorithm 6 in terms of the same node to eliminate expensive MMIOs.
- **Outstanding Requests (a.k.a. OR).** Usually, RDMA NIC (RNIC) executes one-sided verbs sequentially, meaning that a verb starts to be sent until the re-

Table 3 A comparison of network types for YCSB workload

Algorithm	TCP baseline		+two-sided		+one-sided		
	tps(k)	$\overline{\mathcal{U}^e_{CPU}}$	tps(k)	$\overline{\mathcal{U}^e_{CPU}}$	tps(k)	$\overline{\mathcal{U}^e_{CPU}}$	σ
No-Wait	26.74	1.6 %	85.36	6.2 %	**991.07(37.1X)**	15.2 %	23.5
Wait-Die	31.03	2.1 %	85.64	7.2 %	**807.78(26.0X)**	13.7 %	30.2
Wound-Wait	27.98	1.9 %	85.02	6.5 %	**714.57(25.5X)**	15.6 %	31.2
T/O	32.10	2.3 %	85.66	6.9 %	**853.63(26.6X)**	17.5%	25.4
MVCC	32.34	2.3 %	84.84	7.2 %	**939.12(29.0X)**	17.9 %	22.8
Silo	25.01	1.8 %	69.57	5.5 %	**1071.59(42.8X)**	18.8 %	17.7
MaaT	25.58	3.2 %	49.40	10.2 %	**460.68(18.0X)**	14.3 %	51.3
Cicada	27.11	2.1 %	82.62	9.5 %	**720.66(26.6X)**	15.4 %	32.2
Calvin	249.62	2.0 %	261.26	2.0 %	**207.78(0.8X)**	2.0 %	

sults of previous verbs return. To improve the degree of concurrency, OR optimizes the mechanism of message communication by starting to send the verb upon the accomplishment of sending the previous one. In our framework, based on DB, we further optimize the batches to be sent to different nodes using OR.

- **Passive Ack (a.k.a. PA).** In our framework, RNIC processes verbs in a first-come-first-serve manner with a reliable connection. Therefore, given a batch of verbs, as long as we acknowledge the result of the last verb, we can guarantee that the results of previous verbs have also been returned. By doing this, redundant acknowledgment for previous verbs can be eliminated and hence save the bandwidth of RNIC. In our framework, based on DB, we further optimize multiple batches to be sent using PA.

7 Experiment

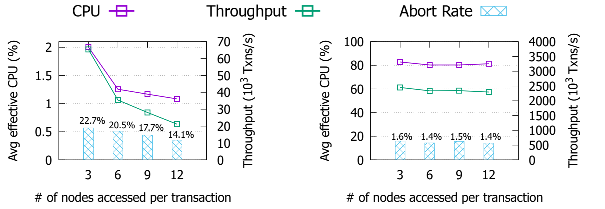
7.1 Setup

We use two popular OLTP benchmarks, YCSB [17] and TPCC [57], to evaluate our re-implementations of concurrency control algorithms.

YCSB [17] is a comprehensive benchmark that simulates large-scale Internet applications. Its dataset contains a single 10-column relation, in which each tuple occupies 1KB. The table is horizontally partitioned, and each node is set to have a fixed number of 10 million records, resulting in 10GB of data per node. Each transaction of the workloads is set to have a fixed number of 10 read/write operations that access data items following the Zipfian distribution. YCSB provides adjustable parameters to simulate workloads with diverse characteristics. The skew factor parameter determines the degree of contention where the access of records follows the Zipfian distribution. The write-ratio parameter controls the ratio of write operations in transactions. Setting the write ratio to 0.2 means that there are 80% reads and 20% writes among all transactions. In this scenario, there might be approximately 10% ($0.8^{10} \approx 0.1$) read-only transactions in the system, while the remaining 90% could be mixed read/write transactions. Additionally, YCSB offers the (θ) parameter, which enables controlling the number of data nodes that will be accessed per transaction. **By default, we set both the write ratio and the skew factor to 0.2, and set θ to 2.**

TPCC [57] is another OLTP benchmark that simulates warehouse ordering applications. Its dataset contains 9 relations, and each warehouse is set to have 100MB of data. By default, we set the number of warehouses per node to 32. TPCC contains 5 types of transactions, among which Payment, New-order, and Delivery are read-write transactions, Stock-level and Order-status are read-only transactions. As Delivery, Stock-level and Order-status only involve local operations, similar to previous works [31,67], we focus on Payment and New-order only to evaluate the transaction scalability of distributed transactions.

We evaluate our system on 4 machines of an RDMA-capable EDR cluster. Each node is equipped with one Intel(R) Xeon(R) Gold 5220 CPU @ 2.20GHz (18 cores×2 HT) processor, 128GB RAM, and one ConnectX-5 EDR 100Gb/s InfiniBand MT27800. By default, each machine is assigned one RCBench node and one client. Each RCBench node is configured to have 4 threads to receive the transactions and send the response from/to clients, and 24 threads to execute transactions, each thread is set to have 8 coroutines. Each client is configured to have 4 threads to generate new transactions, and 4 threads to send the transactions and receive the response to/from RCBench nodes. For each experiment, we initiate a 30-second warm-up period followed by the collection of results for the subsequent 60 seconds.



(a) TCP/IP network (b) RDMA-capable network

Fig. 11 The throughput follows the same trend of $\overline{\mathcal{U}^e_{CPU}}$

7.2 Effective CPU Utilization Rate

In order to evaluate the efficiency of RCBench, we propose a metric in this section to assess the extent to which our designs take full advantage of RDMA. Specifically, by noticing that the introduction of RDMA significantly reduces the network overhead, it is widely recognized that the bottleneck has shifted from network to CPU in RDMA-capable distributed clusters. The CPU utilization rate appears to be a reasonable metric in this context. However, the existing formulation of CPU utilization rate generally incorporates irrelevant overheads, such as the thread/coroutine scheduling costs, etc. As a result, they do not perform as a precise indicator. To address this issue, we propose a new metric called "effective CPU utilization rate ($\mathcal{U}^e_{CPU}$)" to measure the actual utilization of the CPU. The formula is shown below.

$$\mathcal{U}^e_{CPU} = \frac{\text{Effective time to execute committed transactions}}{\text{Total time to execute transactions}}$$

$\mathcal{U}^e_{CPU}$ is defined as the proportion of total CPU execution time spent on the execution of read/write and concurrent control operations of committed transactions. Specifically, time spent on network communication, thread/coroutine coordination, and idle time is excluded from the calculation of $\mathcal{U}^e_{CPU}$. By tracing, breaking down, and averaging the time spent on each operation in threads, we can measure $\mathcal{U}^e_{CPU}$ and use it to assess the effectiveness of our re-implementations. The higher $\mathcal{U}^e_{CPU}$ is, the better RDMA is leveraged for efficient transaction processing. After calculating $\mathcal{U}^e_{CPU}$ for each node, we determine the average $\mathcal{U}^e_{CPU}$ of $\mathcal{N}$ nodes, denoted as $\overline{\mathcal{U}^e_{CPU}}$, for the entire distributed system. Formally, $\overline{\mathcal{U}^e_{CPU}}$ is calculated as follows:

$$\overline{\mathcal{U}^e_{CPU}} = \frac{\sum_{i=1}^{\mathcal{N}} (\mathcal{U}^e_{CPU}\ of\ node\ i)}{\mathcal{N}}$$

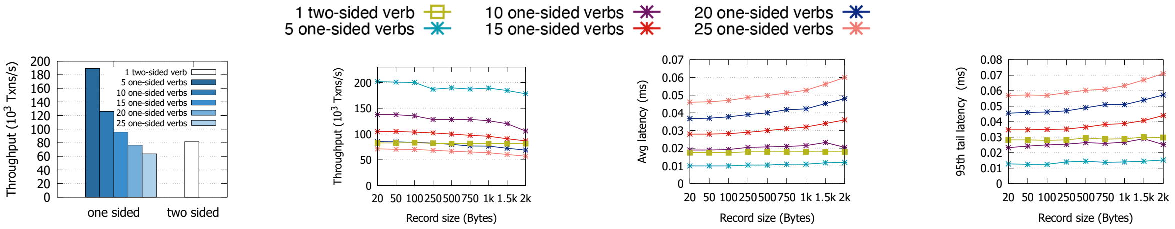
$\overline{\mathcal{U}^e_{CPU}}$ can be interpreted as the effective working time per unit of time to execute committed transactions, which is similar to throughput quantified by the number of committed transactions per unit of time. Figure 11(a) shows the throughput, $\overline{\mathcal{U}^e_{CPU}}$ and abort rate of 2PL algorithm in conventional shared-nothing TCP/IP architecture. As the number of accessed data nodes per transaction increases from 3 to 12, the $\overline{\mathcal{U}^e_{CPU}}$ follows the same trend as that of the throughput, verifying it as a reliable metric of our re-implementation efficiency. Moreover, the abort rate decreases as the throughput decreases, indicating that the drop in performance is not attributable to aborts. For comparison, we plot the results of RCBench under the same setup in Figure 11(b), which displays a much stabler throughput, $\overline{\mathcal{U}^e_{CPU}}$, and abort rate.

Note that the CPU utilization metric proposed by Binnig et al. [10,68] is somewhat similar to our proposed $\overline{\mathcal{U}^e_{CPU}}$. However, they mainly focus on CPU cycles consumed by network communications itself. Instead, it is considered ineffective and excluded from $\overline{\mathcal{U}^e_{CPU}}$. Harding [31] discusses the breakdown of distributed transaction executions in detail. However, they do not identify ineffective works, and abstract no concept concerning CPU utilization rate, which is different from our idea of $\overline{\mathcal{U}^e_{CPU}}$.

7.3 Comparison between RDMA Verbs

We investigate the efficiency difference between one-sided and two-sided verb invocations. We assume that executing a transaction requires either one two-sided verb or 5, 10, 15, 20, or 25 one-sided verbs, with each verb accessing a fixed length of data, such as 1500 bytes. This is because some algorithms, like Silo, require reading the metadata of the same data items multiple times to check if it has been modified by other transactions. To ensure fairness, both one-sided and two-sided verbs are called sequentially, that is, another one/two-sided verb cannot be called until the result of the previous message is returned.

As shown in Figure 12(a), we plotted the throughput of transactions that required either one two-sided verb or 5, 10, 15, 20, or 25 one-sided verbs. The graph shows that when the data access size is fixed at 1500 bytes, the throughput of one two-sided verb transaction is equivalent to using 20 one-sided verbs transaction. To further evaluate the performance of one-sided and two-sided verbs, we plotted the latency and tail (95th) latency of the transactions in Figures 12(c) and 12(d), respectively. It can be observed that the latency and tail latency of one two-sided verb transaction (yellow lines) are equivalent to that of using 10 one-sided verbs transaction. Even when we adjust the data access size from 20 to 2000, as shown in Figures 12(b), 12(c), and 12(d), the difference in throughput and latency between using one-sided verbs and two-sided verbs remains consistent.

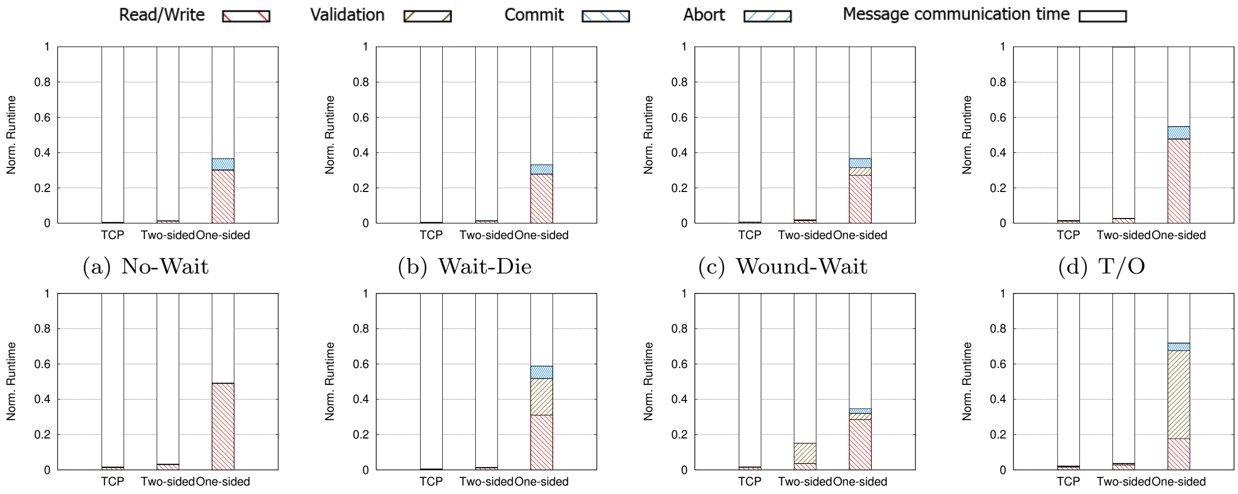


(a) Throughput (1.5KB) (b) Throughput (c) Average latency (d) 95th latency

Fig. 12 A performance comparison between RDMA verbs.

Table 4 A comparison of network types for YCSB workload

Algorithm	TCP baseline			+two-sided			+one-sided			
	TPS(k)	$\overline{\mathcal{U}^e_{CPU}}$	AR	TPS(k)	$\overline{\mathcal{U}^e_{CPU}}$	AR	TPS(k)	$\overline{\mathcal{U}^e_{CPU}}$	AR	σ
No-Wait	26.74	1.6 %	20.8 %	85.36	6.2 %	15.4 %	**991.07(37.1X)**	15.2 %	0.0 %	23.5
Wait-Die	31.03	2.1 %	13.4 %	85.64	7.2 %	7.8 %	**807.78(26.0X)**	13.7 %	0.8 %	30.2
Wound-Wait	27.98	1.9 %	6.9 %	85.02	6.5 %	10.3 %	**714.57(25.5X)**	15.6 %	0.0 %	31.2
T/O	32.10	2.3 %	1.4 %	85.66	6.9 %	1.4 %	**853.63(26.6X)**	17.5 %	0.5 %	25.4
MVCC	32.34	2.3 %	0.9 %	84.84	7.2 %	1.2 %	**939.12(29.0X)**	17.9 %	0.3 %	22.8
Silo	25.01	1.8 %	17.9 %	69.57	5.5 %	17.5 %	**1071.59(42.8X)**	18.8 %	0.3 %	17.7
MaaT	25.58	3.2 %	7.2 %	49.40	10.2 %	10.0 %	**460.68(18.0X)**	14.3 %	0.0 %	51.3
Cicada	27.11	2.1 %	3.5 %	82.62	9.5 %	2.4 %	**720.66(26.6X)**	15.4 %	0.3 %	32.2
Calvin	249.62	2.0 %	0.0 %	261.26	2.0 %	0.0 %	**207.78(0.8X)**	2.0 %	0.0 %	



(a) No-Wait (b) Wait-Die (c) Wound-Wait (d) T/O

(e) MVCC (f) Silo (g) MaaT (h) Cicada

Fig. 13 Time breakdown under different networks.

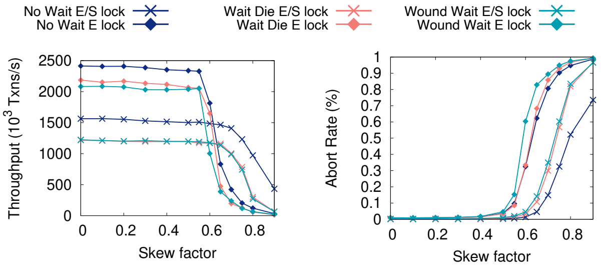
7.4 Effect of RDMA Networks

We study the effect of RDMA networks on YCSB in terms of throughput (TPS), $\overline{\mathcal{U}^e_{CPU}}$, and abort rate (AR). For illustration, the re-implementations over TCP/IP Ethernet networks, two-sided RDMA networks, and one-sided RDMA networks are referred to as *TCP/IP, two-sided, one-sided*. Note, *two-sided* is simply implemented by replacing the network invocations in Deneva [31] with two-sided verb invocations.

The results are reported in Table 4. As we can see, for *TCP/IP*, $\overline{\mathcal{U}^e_{CPU}}$ is rather low, ranging only from 1.6%-3.2%, leading to a poor throughput; for two-sided, compared with TCP/IP, except Calvin, $\overline{\mathcal{U}^e_{CPU}}$ improves by at least 3.0X, causing an improvement of throughput ranging from 1.9X to 3.2X; compared with *two-sided*, *one-sided* further improves $\overline{\mathcal{U}^e_{CPU}}$ and throughput ranging from 1.4X to 3.4X and 8.4X to 11.6X, respectively. This is because, in our framework, we completely eliminate the expensive 2PC overhead and enjoy all benefits of RDMA networks. Compared with *TCP/IP*, *one-sided* achieves a significant improvement of throughput ranging from 18.0X to 42.8X except for Calvin.

To precisely analyze the reason why *one-sided* outperforms *TCP/IP* and *two-sided*, as shown in Figure 13, we evaluate the time breakdown for the eight algorithms except for Calvin. We break down the execution time of a transaction into five parts, including the read/write operations, the validation operations, the commit operations, the abort operations, and the message communication time in all three phases. As we can see, for each algorithm, *TCP/IP* and *two-sided* consume almost all of the time to perform message communication, while *one-sided* uses 28% (Cicada) – 67% (Wait-Die) of the time to perform remote one-sided verbs. While the two-sided verb benefits from the high-performance InfiniBand network, its message communication time is still relatively long due to the additional overhead incurred by the communication mechanism. This includes operations such as message sending and receiving, thread/coroutine scheduling before executing the message, and the creation of the transaction context. These operations cannot be avoided by two-sided verbs and contribute to the overall communication overhead. Thus, *one-sided* achieves such a high speedup over the *TCP/IP* and *two-sided* variants. In Table 4, we can observe that the *one-sided* variants have a much lower abort rate (AR) than the *TCP/IP* and *two-sided* variants. This is because *one-sided* variants have lower message communication time overhead, lower transaction processing time, and lower concurrency conflicts with other transactions, making them more efficient and less prone to conflicts and aborts.

In addition, for the throughput of different algorithms under *one-sided*, Silo performs the best. The reason is that the throughput is closely related to the averaged number (σ) of primitive invocations per transaction. For reference, we report σ for each algorithm in Table 4, showing Silo takes the smallest σ. Often, a smaller σ leads to a higher throughput. However, the complexity of concurrency control algorithms may slightly affect the performance. For example, compared with No-Wait, MVCC takes a slightly smaller σ but has a lower throughput due to its complexity of traversing versions. Note that in the current YCSB workload, each transaction consists of 10 operations and accesses two nodes, which means about half of the operations (approximately 5 operations) may be remote operations. In Table 4, σ represents the total number of primitive invocations for approximately 5 remote read/write, 1 validation, and 1 commit operation of a transaction. For *two-sided*, the same workload requires 7 rounds of two-sided network communication (5 remote read/write + 1 validation + 1 commit operations), which can be translated to about 70–140 one-sided verb invocations, as described in Section 2.2 and 7.3. Moreover, σ is not expected to increase significantly in the TPCC workload since a New-order transaction has 5–15 remote read/write operations, and a Payment transaction has only 2 remote read/write operations. As opposed to the other concurrency control algorithms, Calvin takes a similar throughput under both TCP/IP and RDMA networks. This is because Calvin is bottlenecked by its scheduler rather than the networks, while RDMA is only used to alleviate the bottleneck by the networks. In the rest of the paper, we focus on one-sided and do not report the results for Calvin.



(a) TPS

(b) AR

Fig. 14 Effect of different lock types.

7.5 Comparison with Hybrid Variants

Following the hybrid approach in DrTM+H [62], we study the throughput under the combination of one-sided and two-sided RDMA verbs, as shown in Table 5. Because the lock-based algorithms and the timestamp-based algorithms do not require the validation phase, we implement two extra scenarios for these algorithms, only using one-sided verbs in the read/write phase (r/w in Table 5) or the commit phase (co in Table 5). For optimistic algorithms, we re-implement six scenarios for them, using one-sided verbs in (1) the read/write phase (r/w), (2) the validation phase (va), (3) the commit phase (co), (4) the read/write and validation phase (r/w+va), (5) the validation and commit phase (va+co), and (6) the read/write and commit phase (r/w+co).

As we can see, by introducing one-sided verbs at some phases, the throughput of most algorithms can be improved but is still not comparable to that using one-sided verbs at all phases (all in Table 5). The reason is that the hybrid variants still rely on two-sided verbs, which are subject to the overhead of remote node participation and scheduling, thus affecting performance. Therefore, in RCBench, one-sided verbs are more suitable for promoting each algorithm.

Table 5 A comparison of one-sided and two-sided RDMA verbs combinations

<table>
<tr><th>Algorithm</th><th>two-sided</th><th colspan="3">r/w</th><th colspan="3">co</th><th>all</th></tr>
<tr><td>No-Wait</td><td>85.36</td><td colspan="3">359.20</td><td colspan="3">113.22</td><td>991.07</td></tr>
<tr><td>Wait-Die</td><td>85.64</td><td colspan="3">387.28</td><td colspan="3">139.39</td><td>807.78</td></tr>
<tr><td>Wound-Wait</td><td>85.02</td><td colspan="3">384.48</td><td colspan="3">110.39</td><td>714.57</td></tr>
<tr><td>T/O</td><td>85.66</td><td colspan="3">106.98</td><td colspan="3">96.14</td><td>853.63</td></tr>
<tr><td>MVCC</td><td>84.84</td><td colspan="3">236.08</td><td colspan="3">103.25</td><td>939.12</td></tr>
<tr><th></th><th>two-sided</th><th>r/w</th><th>va</th><th>co</th><th>r/w+va</th><th>va+co</th><th>r/w+co</th><th>all</th></tr>
<tr><td>Silo</td><td>69.57</td><td>235.06</td><td>199.38</td><td>196.74</td><td>143.25</td><td>139.96</td><td>379.65</td><td>1071.59</td></tr>
<tr><td>MaaT</td><td>49.40</td><td>66.84</td><td>100.60</td><td>71.76</td><td>228.83</td><td>141.79</td><td>151.63</td><td>460.68</td></tr>
<tr><td>Cicada</td><td>82.62</td><td>163.87</td><td>126.83</td><td>126.04</td><td>345.63</td><td>169.23</td><td>215.82</td><td>720.66</td></tr>
</table>

Table 6 A comparison of one-sided implementation with different optimization

Algorithm	one-sided		+ db&pa&or		+ coroutine		+ cor+db&pa&or	
	TPS(k)	$\overline{\mathcal{U}^{e}_{CPU}}$	TPS(k)	$\overline{\mathcal{U}^{e}_{CPU}}$	TPS(k)	$\overline{\mathcal{U}^{e}_{CPU}}$	TPS(k)	$\overline{\mathcal{U}^{e}_{CPU}}$
No-Wait	991.1	15 %	1091.0	21 %	**2274.3**	78 %	2145.6	58 %
Wait-Die	807.8	14 %	928.9	20 %	**1909.7**	78 %	1749.6	60 %
Wound-Wait	714.5	16 %	907.7	19 %	**1803.1**	75 %	1771.7	60 %
T/O	853.7	17 %	887.7	18 %	1663.3	50 %	**1706.3**	50 %
MVCC	939.1	17 %	970.0	17 %	1839.6	48 %	**1865.0**	51 %
Silo	1071.5	19 %	1184.3	21 %	2383.4	73 %	**2385.0**	75 %
MaaT	460.7	14 %	531.9	15 %	**796.7**	42 %	791.7	37 %
Cicada	720.7	15 %	759.9	15 %	1492.3	54 %	**1506.0**	55 %

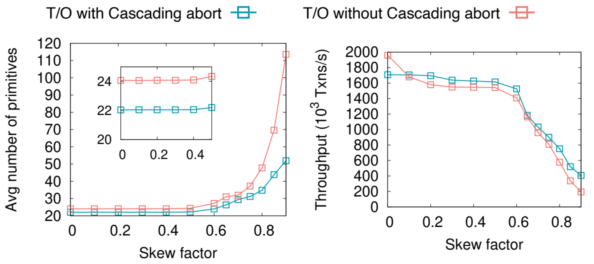
7.6 Effect of Various Optimizations

7.6.1 General Optimizations

We investigate the effect of applying optimizations including coroutine, OR, DB, and PA. Table 6 reports the throughput and $\overline{\mathcal{U}^{e}_{CPU}}$ by combining different optimizations. As we can see, by introducing DB, PA, and OR, $\overline{\mathcal{U}^{e}_{CPU}}$ and the throughput are improved slightly ranging from 1X to 1.47X and 1.03X to 1.27X, respectively; interestingly, by introducing coroutine only, both $\overline{\mathcal{U}^{e}_{CPU}}$ and throughput are improved greatly, indicating that coroutine is a more important optimization factor than the others; adding DB, PA, and OR with coroutine can only improve $\overline{\mathcal{U}^{e}_{CPU}}$ and throughput of some algorithms, such as T/O and Cicada, by at most 2%, while cannot bring benefits for the others. This is because the coroutine reduces the idle time of the CPU, which dominates the benefit brought by DB, PA, and OR. Again, Silo performs the best, followed by No-Wait and MVCC.

7.6.2 Variants of 2PL

We re-implement and compare variants of 2PL under two types of locking mechanisms: exclusive/shared locking (abbreviated as *E/S lock*) and exclusive locking (abbreviated as *E lock*). The results are reported in Figure 14. When the skew factor ranged from 0 to 0.6, variants under *E lock* outperformed those under *E/S lock*. This is because *E lock* used fewer one-sided verb invocations than *E/S lock* to acquire locks. In a low contention scenario, the benefit of improving concurrency was outweighed by the overhead of conflict examination. However, when the skew factor exceeded 0.7, the abort rate of *E lock* increased more quickly than that of *E/S lock*. This indicated that the benefit of improving concurrency became comparable to, or even outweighed, the overhead of conflict examination. In this scenario, *E/S lock* performed better than *E lock*.



(a) Number of primitive invocations

(b) TPS

Fig. 15 Effect with or without cascading abort.

7.6.3 Variants of T/O

We plot the result of T/O with or without enabling cascading abort in Figure 15. As we can see, T/O with or without cascading abort performs similarly, and in most cases, as opposed to the phenomenon in the centralized environment, T/O with cascading abort performs slightly better. This is because, as shown in Figure 15(a), T/O without cascading abort consumes extra one-sided verb invocations to manipulate two pending lists ($X^m.prL$ and $X^m.pwL$), which is comparable to that brought by the cascading abort.

7.7 Effect of Contention Levels

We evaluate the performance by varying the skew factor from 0 to 0.95. The throughput and $\overline{\mathcal{U}^e_{CPU}}$ under the low write-ratio (write-ratio = 0.2) are reported in Figure 16(b) and 16(a), respectively. As we can see, the throughput and $\overline{\mathcal{U}^e_{CPU}}$ of all algorithms remain relatively stable when the skew factor varies from 0 to 0.6. However, when the skew factor is greater than 0.65, the performance of all algorithms drops sharply. The optimal choice of the algorithm also tends to change with the skew factor. While Silo excels under the low and moderate contention (i.e. skew factor less than 0.8) due to the same reason mentioned in Section 7.6, Cicada performs the best under high contention, mostly attributable to the multi-version mechanism it uses to enable concurrent read and write operations. Figure 16(d) and 16(c) report the throughput and $\overline{\mathcal{U}^e_{CPU}}$ under the high write-ratio (write-ratio = 0.8), respectively. The results follow a similar trend to those under low write ratios.

7.8 Effect of Write Ratios

We evaluate the performance by varying write ratios from 0 to 1.0. Figure 17(b) and 17(a) report the throughput and $\overline{\mathcal{U}^e_{CPU}}$ under low contention (skew factor = 0.2), respectively. It can be observed that with increasing the write ratio, the throughput of Silo, Cicada, MVCC, and T/O drops slightly. Because each read or write operation takes the same number of primitive calls and acquires exclusive lock in No-Wait, Wait-Die, Wound-Wait, and MaaT, the throughput of them remains stable. We then report the results under high contention (skew factor = 0.8) in Figure 17(d) and 17(c), respectively. Due to the same reason, the throughput and $\overline{\mathcal{U}^e_{CPU}}$ of No-Wait, Wait-Die, Wound-Wait, and MaaT remain stable. However, the throughput and $\overline{\mathcal{U}^e_{CPU}}$ of Silo, Cicada, MVCC, and T/O drop significantly with increasing the write ratio due to the high contention level. Note that due to the multi-version mechanism, Cicada performs the best when the write ratio is larger than 0.2.

7.9 Scalability

Scalability is evaluated from two perspectives. First, we measure transaction scalability by varying the number (θ) of data nodes to be accessed per transaction while leaving the total number ($\mathcal{N}$) of data nodes in the system to be constant. In contrast, system scalability is measured with a fixed θ and varied $\mathcal{N}$. To fix θ, we follow the implementation in Deneva, where the client knows the storage locations of all data items based on their hash value and generates transactions with fixed θ based on the storage locations of all data items.

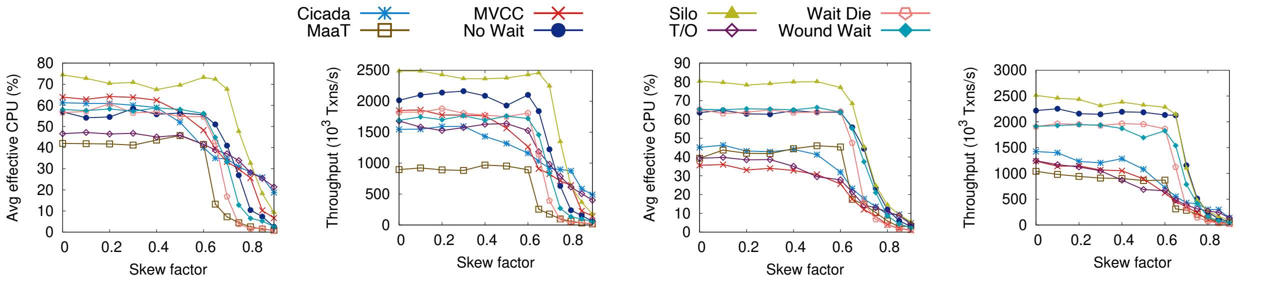
7.9.1 Transaction Scalability

We test the transaction scalability by varying θ from 3 to 12, and fixing $\mathcal{N}$ to 12. Each transaction is composed of 12 read/write operations, with the percentage of remote operations fixed to 11/12. We report the throughput, total CPU utilization rate ($\Sigma = \mathcal{N} \times \overline{\mathcal{U}^e_{CPU}}$), and abort rate (AR) under low contention (skew factor = 0.2) in Figure 18(b), 18(a), and 18(c), respectively. The throughput for different algorithms ranges from 1.0M TPS to 2.4M TPS, and for all algorithms, Σ as well as the throughput drops rather slightly. We report the results under high contention (skew factor = 0.8) in Figure 18(e), 18(d), and , and 18(f), respectively. For the same reason, Σ and the throughput follow the same trend as those under low contention. This finding verifies that our re-implementations of algorithms using RDMA networks demonstrate arguably transaction scalability.

Note, the throughput in this experiment cannot be directly compared with those of other experiments, for two reasons: first, the number of YCSB operations is set to 12 in this experiment, while it is set to 10 in other experiments; second, while the transactions in this experiment involve 11 remote operations, the transactions in the other experiments typically involve around 5 remote operations.

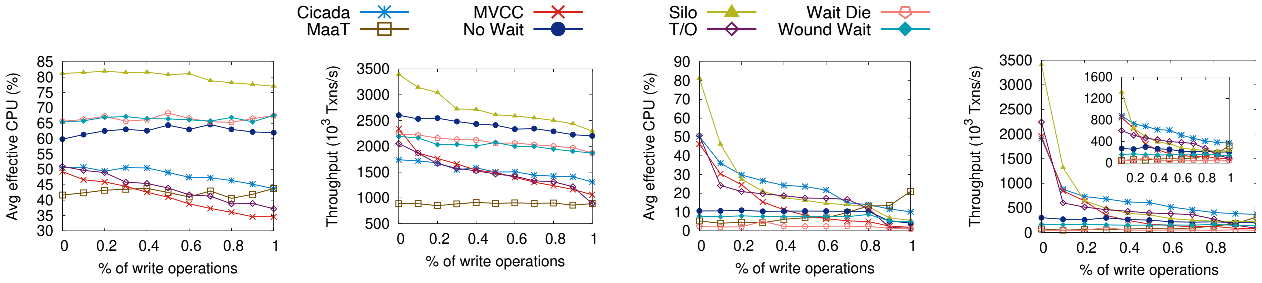
7.9.2 System Scalability

We evaluate the system scalability by varying $\mathcal{N}$ from 3 to 12 and fixing θ to 2. We report Σ under read-only workload, moderate contention (skew factor = 0.5) workload and high contention (skew factor = 0.8) workload in Figure 19(a), 19(c) and 19(e), respectively. As we can see, when $\mathcal{N}$ varies, Σ increases linearly. Besides, for the same $\mathcal{N}$, Σ drops slightly when the contention increases. We also plot the throughput in Figure 19(b), 19(d) and 19(f), respectively. The results show that the throughput increase linearly, which follows the



(a) $\overline{\mathcal{U}^e_{CPU}}$(low write-ratio) (b) TPS(low write-ratio) (c) $\overline{\mathcal{U}^e_{CPU}}$(high write-ratio) (d) TPS(high write-ratio)

Fig. 16 The comparison of different contention levels.



(a) $\overline{\mathcal{U}^e_{CPU}}$(low contention) (b) TPS(low contention) (c) $\overline{\mathcal{U}^e_{CPU}}$(high contention) (d) TPS(high contention)

Fig. 17 The comparison of different write-ratios.

same trend with Σ. We plot the results over TPCC in Figure 20(a)–20(d). As we can see, the result follows a similar trend to that over YCSB.

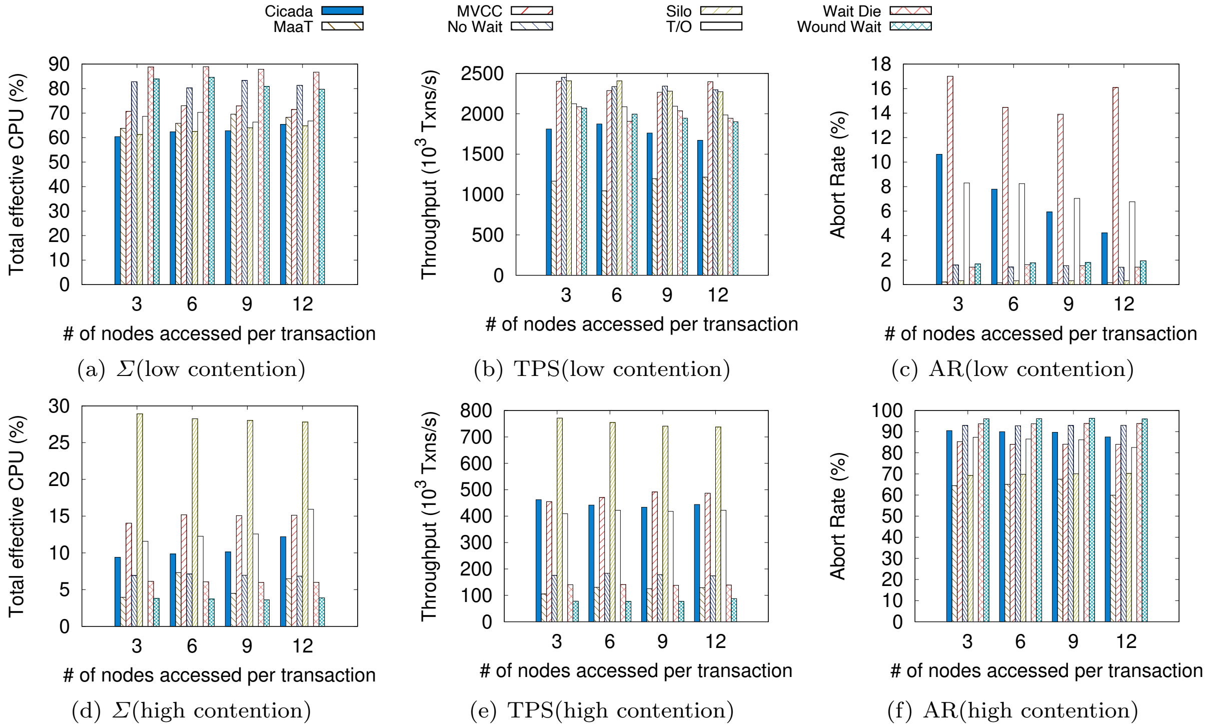
7.10 Comparison with Other RDMA-based Algorithms

To show the effectiveness of our re-implementations, we compare Silo against DrTM+H [62], another RDMA-based re-implementation of Silo, and compare No Wait, Wait Die, and Wound Wait against DSLR[66], another RDMA-based re-implementation of 2PL variants. We report the comparison in Figure 21. It can be observed that our Silo outperforms DrTM+H in terms of both throughput and $\overline{\mathcal{U}^e_{CPU}}$ significantly. Besides, there is an obvious trend of growing returns when the number of threads increases. This is because, in DrTM+H, the remote accesses in the validation phase and the commit phase are all implemented using two-sided verbs, while in our work, we optimize all the remote accesses completely using one-sided verbs, thus improving $\overline{\mathcal{U}^e_{CPU}}$ and the throughput significantly. In addition, as we can see, our No-Wait under *E/S lock* mechanism performs better than DSLR, and our Wait-Die and Wound-Wait take a similar throughput as DSLR. This is because DSLR consumes a similar number of one-sided verb invocations as Wait-Die and Wound-Wait, while No-Wait consumes the smallest one-sided verb invocations. However, the throughput and $\overline{\mathcal{U}^e_{CPU}}$ of Wait-Die and Wound-Wait drop significantly when the number of threads is greater than 32. This is because Wait-Die and Wound-Wait maintain a fixed size of $X^m.pL$, which causes extra aborts of transactions under the high contention levels. Additionally, we evaluated the transaction scalability of the algorithms in RCBench and compared them with the DrTM+H system, as shown in Figure 22. We observed that although both the algorithms re-implemented in RCBench and DrTM+H can achieve transactional scalability, RCBench ensures up to 1.1X better transaction scalability than DrTM+H. Moreover, the throughput of concurrency control algorithms in RCBench, such as Silo and No-Wait, is higher than that in DrTM+H.

7.11 Summary

After conducting extensive evaluations, we summarize the major experimental findings below.

1. Transaction scalability can be arguably achieved (Section 7.9.1).
2. Concurrency control algorithms can achieve system scalability. (Section 7.9.2).
3. For the same algorithm, $\mathcal{U}^e_{CPU}$ is the dominant factor to scale out distributed transaction processing (Section 1 and 7.9).
4. It is practical and convenient to re-implement concurrency control algorithms using our proposed primitives, and even performs better than some customized implementations in many cases (Section 7.10).



(a) Σ(low contention)

(b) TPS(low contention)

(c) AR(low contention)

(d) Σ(high contention)

(e) TPS(high contention)

(f) AR(high contention)

Fig. 18 Transaction scalability.

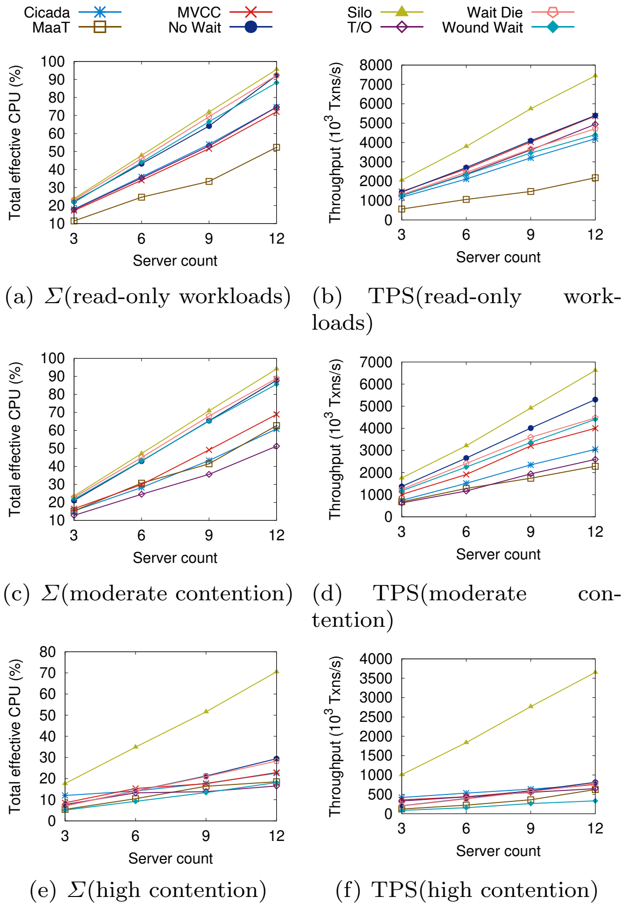
5. Optimizing Calvin using RDMA cannot bring obvious benefits (Section 6.2.4 and 7.4). For other algorithms, RDMA networks bring significant performance improvement, and the degree of improvement closely relies on the number of primitive invocations and metadata complexity. Based on these criteria, Silo is reported to be the best in most cases (Section 7.4, 7.7 and 7.8).
6. Among all optimization techniques, coroutine brings the maximal performance improvement (1.7X to 2.5X). For 2PL variants, single exclusive locking is preferred in moderate or low contention scenarios, while exclusive/shared locking is preferred in high contention scenarios (Section 7.6.1 and 7.6.2).
7. Optimizations in the centralized environment may not work well in RDMA networks due to a prohibitive number of primitive invocations, e.g., RDMA-T/O (Section 7.6.3).

8 Related Work

To the best of our knowledge, this is the first to comprehensively evaluate the transaction scalability of concurrency control algorithms using RDMA-based primitives. Our study is related to previous works on 1) transaction scalability evaluations of concurrency control algorithms over TCP/IP networks and 2) optimizations for them using RDMA networks.

We have witnessed a wide spectrum of concurrency control algorithms to guarantee the serializable isolation level. Because there does not exist a single algorithm that can perform the best in all scenarios, figuring out the best algorithm for a specific application scenario naturally becomes an important problem. A few works [3,9,12,56,54,13] make theoretical analyses over the benefits of different kinds of algorithms. More works are proposed to put concurrency control algorithms in the same centralized framework to do evaluations [35, 33,24,67,15,64,53,34]. Recently, there is an increasing interest in evaluating algorithms for distributed transaction processing [12,14,31]. The results show that these algorithms cannot achieve transaction scalability due to the limitations of slow network and coordination overhead. There are also several works [25,19,23,4,41,18, 36,48,50,70,55,20,61] that focus on improving the system scalability. The intuitive idea is to transform distributed transactions into local transactions by carefully designing locality-aware partitioning approaches. Yet, static partitioning works only if the optimal data placement is known a priori, while dynamic partitioning often suffers from an expensive data migration overhead.

Using RDMA networks to optimize concurrency control algorithms has become a hotspot in both academia and industry. Most of them are tailored for some particular concurrency control algorithms. Dragojevi et.



(a) Σ(read-only workloads) (b) TPS(read-only workloads)

(c) Σ(moderate contention) (d) TPS(moderate contention)

(e) Σ(high contention) (f) TPS(high contention)

Fig. 19 System scalability under YCSB.

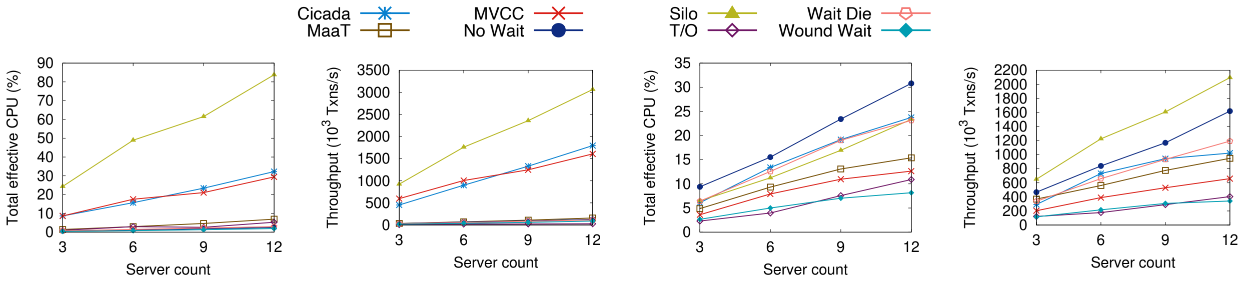
al. [22,21] implement a distributed in-memory database called FaRM and leverage one-sided verbs to optimize Silo. DrTM+H [62] also re-implements Silo and additionally proposes optimizations based on hybrid one-sided and two-sided verbs. NAM-DB [10] optimizes the snapshot isolation algorithm, which can only achieve the snapshot isolation level. To achieve the serializable isolation level in NAM-DB, a complete redesign of the concurrency control algorithm using RDMA is required. A few works [63,16,5,66] concentrate on optimizations of 2PL algorithms. More related to our work, Wang et al. [60] builds a unified framework to re-implement and evaluate the concurrency control algorithms using RDMA. Yet, this framework lacks generality, making each algorithm implemented from scratch independently. Moreover, the evaluation in this framework mainly focuses on the performance with a fixed number the data nodes accessed per transaction. Different from the above works, we focus on transaction scalability under the serializable isolation level. We first explore the dominant factors to scale out distributed transaction processing, and propose a general framework RCBench using one-sided RDMA verbs only, enjoying all benefits of RDMA networks. To enable the implementation of various algorithms, we introduce six primitives and five optimization principles. RCBench utilizes the TCP/IP implementation in Deneva [31] to compare TCP/IP algorithm variants with one-sided algorithm variants. Compared to Deneva, RCBench is built on a shared-memory architecture on top of the RDMA one-sided network, whereas Deneva is a shared-nothing system entirely implemented on the TCP/IP network. In comparison with NAM-DB[10], both RCBench and NAM-DB are built on the RDMA one-sided network. However, RCBench adopts a shared-memory architecture, while NAM-DB uses a compute-storage separation architecture known as network-attached memory. Furthermore, based on RCBench, we provide an additional six primitives and five principles that enable developers to use one-sided RDMA verbs to enhance different algorithms.

9 Conclusions

In this paper, we investigate the problem of whether it is scalable to process distributed transactions using RDMA networks under serializable isolation level. We observe that $\mathcal{U}^e_{CPU}$ is the dominant factor to scale out distributed transactions. To improve $\mathcal{U}^e_{CPU}$, we first propose a framework with six abstracted primitives using one-sided verbs. We then re-implement state-of-the-art concurrency control algorithms with various optimizations simply based on the primitives. Our implementations can enjoy all benefits of RDMA networks. Finally, we conduct a comprehensive experimental study of the transaction scalability of our implementations. We list a few findings that have not yet been reported elsewhere. We are confident that these findings provide a potential guideline to develop highly scalable distributed databases.

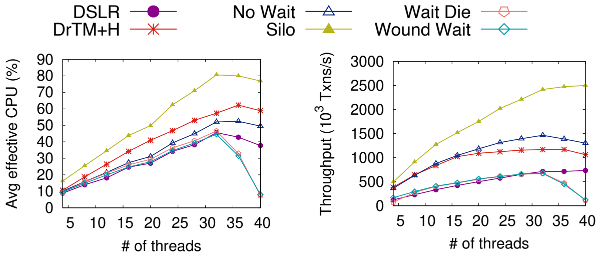
References

1. Michael Abebe, Brad Glasbergen, and Khuzaima Daudjee. Dynamast: Adaptive dynamic mastering for replicated systems. In *36th IEEE International Conference on Data Engineering, ICDE 2020, Dallas, TX, USA, April 20-24, 2020*, pages 1381–1392. IEEE, 2020.
2. Michael Abebe, Brad Glasbergen, and Khuzaima Daudjee. Morphosys: Automatic physical design metamorphosis for distributed database systems. *Proc. VLDB Endow.*, 13(13):3573–3587, oct 2020.
3. Rakesh Agrawal, Michael J. Carey, and Miron Livny. Concurrency control performance modeling: Alternatives and implications. *ACM Trans. Database Syst.*, 12(4):609–654, 1987.
4. Sanjay Agrawal, Vivek R. Narasayya, and Beverly Yang. Integrating vertical and horizontal partitioning into automated physical database design. In *SIGMOD Conference*, pages 359–370. ACM, 2004.



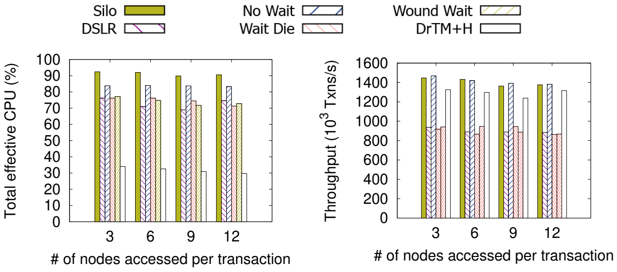
(a) Σ(New-order) (b) TPS(New-order) (c) Σ(Payment) (d) TPS(Payment)

Fig. 20 System scalability under TPCC.



(a) $\overline{\mathcal{U}_{CPU}^{e}}$ (b) TPS

Fig. 21 Comparison with RDMA-based algorithms [62, 66].



(a) Σ(New-order) (b) TPS

Fig. 22 Transaction scalability compared with RDMA-based algorithms.

5. Claude Barthels, Ingo Müller, Konstantin Taranov, Gustavo Alonso, and Torsten Hoefler. Strong consistency is not hard to get: Two-phase locking and two-phase commit on thousands of cores. *Proc. VLDB Endow.*, 12(13):2325–2338, 2019.
6. Hal Berenson, Philip A. Bernstein, Jim Gray, Jim Melton, Elizabeth J. O'Neil, and Patrick E. O'Neil. A critique of ANSI SQL isolation levels. In *SIGMOD Conference*, pages 1–10. ACM Press, 1995.
7. Philip A. Bernstein and Nathan Goodman. Timestamp-based algorithms for concurrency control in distributed database systems. In *VLDB*, pages 285–300. IEEE Computer Society, 1980.
8. Philip A. Bernstein and Nathan Goodman. Concurrency control in distributed database systems. *ACM Comput. Surv.*, 13(2):185–221, 1981.
9. A. Bhide and M. Stonebraker. A performance comparison of two architectures for fast transaction processing. In *International Conference on Data Engineering*, 1988.
10. Carsten Binnig, Andrew Crotty, Alex Galakatos, Tim Kraska, and Erfan Zamanian. The end of slow networks: It's time for a redesign. *Proc. VLDB Endow.*, 9(7):528–539, 2016.
11. Claude Boksenbaum, Michèle Cart, Jean Ferrié, and Jean-François Pons. Certification by intervals of timestamps in distributed database systems. In *VLDB*, 1984.
12. M. J. Carey and M. Livny. Distributed concurrency control performance: A study of algorithms, distribution, and replication. In *Fourteenth International Conference on Very Large Data Bases*, 1988.
13. Michael J. Carey. An abstract model of database concurrency control algorithms. In *SIGMOD Conference*, pages 97–107. ACM Press, 1983.
14. Michael J. Carey and Miron Livny. Parallelism and concurrency control performance in distributed database machines. In *SIGMOD Conference*, pages 122–133. ACM Press, 1989.
15. Michael J. Carey and Waleed A. Muhanna. The performance of multiversion concurrency control algorithms. *ACM Trans. Comput. Syst.*, 4(4):338–378, 1986.
16. Yanzhe Chen, Xingda Wei, Jiaxin Shi, Rong Chen, and Haibo Chen. Fast and general distributed transactions using RDMA and HTM. In *EuroSys*, pages 26:1–26:17. ACM, 2016.
17. Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In *SoCC*, pages 143–154. ACM, 2010.
18. Carlo Curino, Yang Zhang, Evan P. C. Jones, and Samuel Madden. Schism: a workload-driven approach to database replication and partitioning. *Proc. VLDB Endow.*, 3(1):48–57, 2010.
19. Sudipto Das, Shoji Nishimura, Divyakant Agrawal, and Amr El Abbadi. Albatross: Lightweight elasticity in shared storage databases for the cloud using live data migration. *Proc. VLDB Endow.*, 4(8):494–505, 2011.
20. Mohammad Dashti, Sachin Basil John, Amir Shaikhha, and Christoph Koch. Transaction repair for multi-version concurrency control. In *SIGMOD Conference*, pages 235–250. ACM, 2017.
21. Aleksandar Dragojevic, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. Farm: Fast remote memory. In *NSDI*, pages 401–414. USENIX Association, 2014.
22. Aleksandar Dragojevic, Dushyanth Narayanan, Edmund B. Nightingale, Matthew Renzelmann, Alex Shamis, Anirudh Badam, and Miguel Castro. No compromises: distributed transactions with consistency, availability, and performance. In *SOSP*, pages 54–70. ACM, 2015.
23. Aaron J. Elmore, Vaibhav Arora, Rebecca Taft, Andrew Pavlo, Divyakant Agrawal, and Amr El Abbadi. Squall: Fine-grained live reconfiguration for partitioned

main memory databases. In *SIGMOD Conference*, pages 299–313. ACM, 2015.

24. Aaron J. Elmore, Sudipto Das, Divyakant Agrawal, and Amr El Abbadi. Zephyr: live migration in shared nothing databases for elastic cloud platforms. In *SIGMOD Conference*, pages 301–312. ACM, 2011.
25. Aaron J Elmore, Sudipto Das, Divyakant Agrawal, and Amr El Abbadi. Towards an elastic and autonomic multitenant database. In *Proc. of NetDB Workshop*. sn, 2011.
26. Jose M. Faleiro and Daniel J. Abadi. Rethinking serializable multiversion concurrency control. *Proc. VLDB Endow.*, 8(11):1190–1201, 2015.
27. Jose M. Faleiro, Daniel J. Abadi, and Joseph M. Hellerstein. High performance transactions via early write visibility. *Proc. VLDB Endow.*, 10(5):613–624, jan 2017.
28. Jose M. Faleiro, Alexander Thomson, and Daniel J. Abadi. Lazy evaluation of transactions in database systems. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, page 15–26, New York, NY, USA, 2014. Association for Computing Machinery.
29. Jim Gray, Raymond A. Lorie, Gianfranco R. Putzolu, and Irving L. Traiger. Granularity of locks and degrees of consistency in a shared data base. In *Readings in database systems (3rd ed.)*, 1976.
30. Theo Härder. Observations on optimistic concurrency control schemes. *Inf. Syst.*, 9(2):111–120, 1984.
31. Rachael Harding, Dana Van Aken, Andrew Pavlo, and Michael Stonebraker. An evaluation of distributed concurrency control. *PVLDB*, 10(5):553–564, 2017.
32. Ken Higuchi and Tatsuo Tsuji. A linear hashing enabling efficient retrieval for range queries. In *2009 IEEE International Conference on Systems, Man and Cybernetics*, pages 4557–4562, 2009.
33. Jiandong Huang, John A. Stankovic, Krithi Ramamritham, and Donald F. Towsley. Experimental evaluation of real-time optimistic concurrency control schemes. In *VLDB*, pages 35–46. Morgan Kaufmann, 1991.
34. Yihe Huang, William Qian, Eddie Kohler, Barbara Liskov, and Liuba Shrira. Opportunities for optimism in contended main-memory multicore transactions. *Proc. VLDB Endow.*, 13(5):629–642, 2020.
35. Michael J. Jipping and Ray Ford. Predicting performance of concurrency control designs. In *SIGMETRICS*, pages 132–142. ACM, 1987.
36. Evan P. C. Jones. *Fault-tolerant distributed transactions for partitioned OLTP databases.* PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, 2012.
37. Hyeontaek Lim, Michael Kaminsky, and David G. Andersen. Cicada: Dependably fast multi-core in-memory transactions. In *SIGMOD Conference*, pages 21–35. ACM, 2017.
38. Yu-Shan Lin, Ching Tsai, Tz-Yu Lin, Yun-Sheng Chang, and Shan-Hung Wu. Don't look back, look into the future: Prescient data partitioning and migration for deterministic database systems. In *Proceedings of the 2021 International Conference on Management of Data*, SIGMOD '21, page 1156–1168, New York, NY, USA, 2021. Association for Computing Machinery.
39. Yi Lu, Xiangyao Yu, Lei Cao, and Samuel Madden. Aria: A fast and practical deterministic oltp database. *Proc. VLDB Endow.*, 13(12):2047–2060, jul 2020.
40. Christopher Mitchell, Yifeng Geng, and Jinyang Li. Using one-sided RDMA reads to build a fast, cpu-efficient key-value store. In *USENIX Annual Technical Conference*, pages 103–114. USENIX Association, 2013.
41. Andrew Pavlo, Carlo Curino, and Stanley B. Zdonik. Skew-aware automatic database partitioning in shared-nothing, parallel OLTP systems. In *SIGMOD Conference*, pages 61–72. ACM, 2012.
42. Daniel Peng and Frank Dabek. Large-scale incremental processing using distributed transactions and notifications. In *OSDI*, pages 251–264. USENIX Association, 2010.
43. Thamir Qadah, Suyash Gupta, and Mohammad Sadoghi. Q-store: Distributed, multi-partition transactions via queue-oriented execution and communication. In *EDBT*, pages 73–84. OpenProceedings.org, 2020.
44. Thamir M. Qadah and Mohammad Sadoghi. Quecc: A queue-oriented, control-free concurrency architecture. In *Middleware*, pages 13–25. ACM, 2018.
45. Dai Qin, Angela Demke Brown, and Ashvin Goel. Caracal: Contention management with deterministic concurrency control. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 180–194, New York, NY, USA, 2021. Association for Computing Machinery.
46. Abdul Quamar, K. Ashwin Kumar, and Amol Deshpande. SWORD: scalable workload-aware data placement for transactional workloads. In *EDBT*, pages 430–441. ACM, 2013.
47. Daniel J. Rosenkrantz, Richard Edwin Stearns, and Philip M. Lewis II. System level concurrency control for distributed database systems. *ACM Trans. Database Syst.*, 3(2):178–198, 1978.
48. Oliver Schiller, Nazario Cipriani, and Bernhard Mitschang. Prorea: live database migration for multitenant RDBMS with snapshot isolation. In *EDBT*, pages 53–64. ACM, 2013.
49. Marco Serafini, Rebecca Taft, Aaron J. Elmore, Andrew Pavlo, Ashraf Aboulnaga, and Michael Stonebraker. Clay: Fine-grained adaptive partitioning for general database schemas. *Proc. VLDB Endow.*, 10(4):445–456, nov 2016.
50. Jeff Shute, Radek Vingralek, Bart Samwel, Ben Handy, Chad Whipkey, Eric Rollins, Mircea Oancea, Kyle Littlefield, David Menestrina, Stephan Ellner, John Cieslewicz, Ian Rae, Traian Stancescu, and Himani Apte. F1: A distributed SQL database that scales. *Proc. VLDB Endow.*, 6(11):1068–1079, 2013.
51. J.W. Stamos and F. Cristian. A low-cost atomic commit protocol. In *Proceedings Ninth Symposium on Reliable Distributed Systems*, pages 66–75, 1990.
52. Rebecca Taft, Essam Mansour, Marco Serafini, Jennie Duggan, Aaron J. Elmore, Ashraf Aboulnaga, Andrew Pavlo, and Michael Stonebraker. E-store: Fine-grained elastic partitioning for distributed transaction processing systems. *Proc. VLDB Endow.*, 8(3):245–256, nov 2014.
53. Takayuki Tanabe, Takashi Hoshino, Hideyuki Kawashima, and Osamu Tatebe. An analysis of concurrency control protocols for in-memory databases with ccbench. *Proc. VLDB Endow.*, 13:3531 – 3544, 2020.
54. Alexander Thomasian. Concurrency control: Methods, performance, and analysis. *ACM Comput. Surv.*, 30(1):70–119, 1998.
55. Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J. Abadi. Calvin: fast distributed transactions for partitioned database systems. In *SIGMOD Conference*, pages 1–12. ACM, 2012.
56. Bhavani Thuraisingham and Hai-Ping Ko. Concurrency control in trusted database management systems: A survey. *SIGMOD Rec.*, 22(4):52–59, 1993.

57. TPC-C. http://www.tpc.org/tpcc/, 1988.
58. Khai Q. Tran, Jeffrey F. Naughton, Bruhathi Sundarmurthy, and Dimitris Tsirogiannis. Jecb: A join-extension, code-based approach to oltp data partitioning. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, page 39–50, New York, NY, USA, 2014. Association for Computing Machinery.
59. Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. Speedy transactions in multicore in-memory databases. In *SOSP*, pages 18–32. ACM, 2013.
60. Chao Wang and Xuehai Qian. Rdma-enabled concurrency control protocols for transactions in the cloud era. *IEEE Transactions on Cloud Computing*, PP:1–1, 09 2021.
61. Tianzheng Wang and Hideaki Kimura. Mostly-optimistic concurrency control for highly contended dynamic workloads on a thousand cores. *Proc. VLDB Endow.*, 10(2):49–60, 2016.
62. Xingda Wei, Zhiyuan Dong, Rong Chen, and Haibo Chen. Deconstructing rdma-enabled distributed transactions: Hybrid is better! In *OSDI*, pages 233–251. USENIX Association, 2018.
63. Xingda Wei, Jiaxin Shi, Yanzhe Chen, Rong Chen, and Haibo Chen. Fast in-memory transaction processing using RDMA and HTM. In *SOSP*, pages 87–104. ACM, 2015.
64. Yingjun Wu, Joy Arulraj, Jiexi Lin, Ran Xian, and Andrew Pavlo. An empirical evaluation of in-memory multi-version concurrency control. *Proc. VLDB Endow.*, 10(7):781–792, 2017.
65. Chang Yao, Divyakant Agrawal, Gang Chen, Qian Lin, Beng Chin Ooi, Weng-Fai Wong, and Meihui Zhang. Exploiting single-threaded model in multi-core in-memory systems. *IEEE Trans. Knowl. Data Eng.*, 28(10):2635–2650, 2016.
66. Dong Young Yoon, Mosharaf Chowdhury, and Barzan Mozafari. Distributed lock management with RDMA: decentralization without starvation. In *SIGMOD Conference*, pages 1571–1586. ACM, 2018.
67. X. Yu, G. Bezerra, A. Pavlo, S. Devadas, and M. Stonebraker. Staring into the abyss: An evaluation of concurrency control with one thousand cores. *Proc. VLDB Endow.*, 8(3), 2014.
68. Erfan Zamanian, Carsten Binnig, Tim Harris, and Tim Kraska. The end of a myth: Distributed transactions can scale. *Proc. VLDB Endow.*, 10(6):685–696, feb 2017.
69. Erfan Zamanian, Carsten Binnig, and Abdallah Salama. Locality-aware partitioning in parallel database systems. In *SIGMOD Conference*, pages 17–30. ACM, 2015.
70. Zhanhao Zhao. Efficiently supporting adaptive multi-level serializability models in distributed database systems. In *SIGMOD Conference*, pages 2908–2910. ACM, 2021.