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

RCBench: An RDMA-based Framework For Analyzing Concurrency Control Algorithms

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

Received: date / Accepted: date

Abstract Over TCP/IP networks, the performance of processing a distributed transaction that involves an increasing number of data nodes degrades significantly. This phenomenon is called *weak transaction scalability*. Recently, a few attempts have been made to optimize distributed transaction processing by using RDMA-capable high-performance networks. For these works, however, either they are confined to *snapshot isolation level*, lacking applicability for *serializable isolation level* that is the gold standard for concurrency control, or they do not focus on this transaction scalability problem, but instead attempt to improve the performance for some specific concurrency control algorithms. Solving weak transaction scalability under serializable isolation level still remains an open problem. By realizing that there does not exist a single concurrency control algorithm that can perform the best in all cases, in this paper, we first make a thorough investigation of the bottleneck that hinders the scalability of distributed transaction processing; we then propose a unified framework RCBench that is able to alleviate the bottleneck with various optimizations. RCBench is equipped with six abstracted primitives and fully enjoys the benefits of RDMA networks. With these primitives, it is general enough to re-implement concurrency control algorithms and enjoy a fair comparison among the re-implementations in the same framework. We finally make a comprehensive experimental study of the re-implemented algorithms, and the results show that transaction scalability is arguably achieved.

Keywords concurrency control · distributed transaction · scalability · RDMA

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

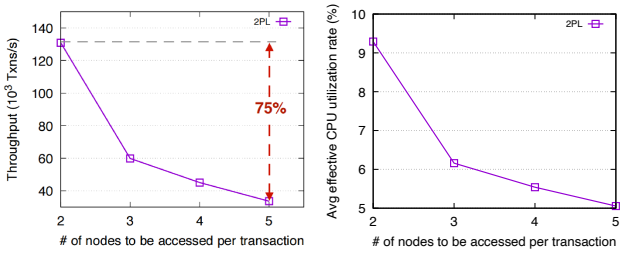
Haixiang Li and Anqun Pan
Tencent Inc., China.
E-mail: {blueseali, aaronpan}@ruc.edu.cn

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

1 Introduction

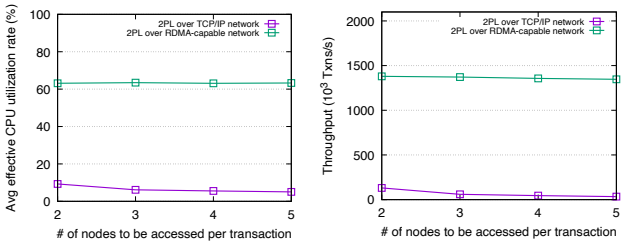
The capability to support distributed transaction processing in database systems is indispensable for many mission-critical applications, such as e-banking and e-commerce. However, it is generally believed that distributed transaction processing over TCP/IP networks cannot scale [53]. That is, the increasing number of data nodes to be involved per transaction makes the system's performance drop significantly. This phenomenon is called **weak transaction scalability**. To show this phenomenon, we implement a two-phase locking concurrency control algorithm (a.k.a. 2PL) on distributed framework Deneva [25] open-sourced by MIT, and conduct an experimental evaluation over YCSB benchmark. We plot the throughput by varying the number of accessed data nodes per transaction in Figure 1(a). Surprisingly, 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.

Thus far, substantial effort has been devoted to alleviating the weak transaction scalability problem. An intuitive idea is to transform distributed transactions



(a) Throughput (b) $\overline{\mathcal{U}^e_{CPU}}$

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



(a) $\overline{\mathcal{U}^e_{CPU}}$ (b) Throughput

Fig. 2 The throughput follows the same trend of $\overline{\mathcal{U}^e_{CPU}}$

into local transactions by carefully designing locality-aware partitioning approaches [20,33]. 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. Note, these legacy approaches are motivated by an assumption that the network is the main bottleneck and consequently local transactions help avoid communication between data nodes.

Recently, the high-performance Infiniband networks with remote direct memory access (a.k.a. RDMA) capability make the network no longer a bottleneck. As a result, a few attempts have been made to optimize distributed transaction processing by using RDMA-capable high-performance networks. These works, however, either are confined to snapshot isolation level, lacking applicability for serializable isolation level that is the gold standard for concurrency control, or do not focus on the transaction scalability problem, but instead attempt to improve the performance for some specific concurrency control algorithms case by case. For example, NAM-DB [53] shows that distributed transaction processing using RDMA can scale, but its technique is confined to snapshot isolation level and cannot be extended to support serializable isolation level due to a different concurrency control mechanism. Other works, attempt to optimize some specific concurrency control algorithms using RDMA under serializable guarantees, such as optimistic concurrency control (OCC) variants [19,47], and 2PL variants [3,14,48,51]. However, these works do not focus on solving the transaction scalability problem. Instead, they optimize the performance by fixing the number of data nodes accessed per transaction. As we will discuss later, solving the transaction scalability problem is quite different from optimizing the performance of concurrency control. That is, the performance may work well when the number of data nodes accessed per transaction is fixed, but degrades significantly when the number of data nodes accessed per transaction increases. Thus far, to the best of our knowledge, whether or not it is able to achieve high transaction scalability under serializable isolation level still remains an open problem.

In this paper, after a careful investigation of the reason(s) why distributed transaction processing under serializability cannot scale, we attempt to build a unified framework over the RDMA networks with the capability to eliminate the bottleneck that hinders the scalability of distributed transaction processing. Based on this framework, we then re-implement mainstream concurrency control algorithms with various optimizations, and make a fair and extensive experimental study to verify whether they are scalable or not. As for the essential reason(s) why distributed transaction processing cannot scale, one of the most cited reasons is the increased contention likelihood [42] that raises the abort rate. On the contrary, Binnig et. al. [8,53,50] considers the contention as only a side effect and the CPU overhead as the essential reason. Yet, the determination of the essential reason(s) is still controversial. To address this issue, we propose a new concept called effective CPU utilization rate ($\mathcal{U}^e_{CPU}$) that is defined as follows.

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

As throughput is quantified by the number of committed transactions per unit of time, we define $\mathcal{U}^e_{CPU}$ to capture a finer-grained description of throughput, and consequently, $\mathcal{U}^e_{CPU}$ can be considered as another way to represent throughput. Note, $\mathcal{U}^e_{CPU}$ excludes ineffective work during the entire execution, such as the execution of aborted transactions, context switching of threads/coroutines, and TCP/IP packing and unpacking in message communication.

We shall show that low $\mathcal{U}^e_{CPU}$ is the essential reason why distributed transaction processing cannot scale. Often, in a shared-nothing distributed database system, a node acts as both a coordinator and a participant. Thus, we collect $\mathcal{U}^e_{CPU}$ for each node and compute their average to capture the overall effective CPU utilization rate of the nodes in the system. Formally, for given $\mathcal{N}$ nodes in the system, we denote the averaged $\mathcal{U}^e_{CPU}$ of

$\mathcal{N}$ nodes as $\overline{\mathcal{U}^e_{CPU}}$, and plot the $\overline{\mathcal{U}^e_{CPU}}$ for Figure 1(a) in Figure 1(b).

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

It can be seen that the throughput follows the same trend of $\overline{\mathcal{U}^e_{CPU}}$ when the number of accessed data nodes per transaction varies. Further, we re-implement 2PL based on our proposed framework, and plot $\overline{\mathcal{U}^e_{CPU}}$ in Figure 2. Again, the throughput follows the same trend of $\overline{\mathcal{U}^e_{CPU}}$. Interestingly, for the re-implementation of 2PL, the throughput remains rather stable when the number of accessed data nodes per transaction varies, further verifying $\mathcal{U}^e_{CPU}$ is the dominant factor to scale out distributed transaction processing.

After identifying the essential reason, i.e., low $\mathcal{U}^e_{CPU}$, that hinders the scalability of distributed transaction processing, we then design a unified framework named RCBench that is able to fully enjoy the benefits of RDMA networks to achieve high $\mathcal{U}^e_{CPU}$ and thus helps achieve scalability. RDMA provides two categories of verbs for programming, one-sided verbs and two-sided verbs. One-sided verbs enjoy all benefits of RDMA but need a heavy re-design of algorithms, while two-sided verbs only enjoy partial benefits of RDMA, but the algorithms do not take much effort to be modified except for the messaging transferring interfaces. Although most RDMA-based implementations employ a hybrid of one-sided and two-sided verbs as a compromise, we choose to completely use one-sided RDMA verbs in RCBench. By doing this, on the one hand, message communication by one-sided RDMA verbs does not need to go through the TCP/IP stack, i.e., one-sided RDMA verbs reduce the invalid CPU utilization on message copying and kernel context switching, and hence improve $\mathcal{U}^e_{CPU}$. On the other hand, one-sided RDMA verbs eliminate the involvement of remote CPU, reduce asynchronous scheduling overhead over remote CPUs, and further improve $\mathcal{U}^e_{CPU}$.

It poses two challenges to build such a unified framework that is able to enjoy all benefits of RDMA networks and facilitate the re-implementation of the concurrency control algorithms.

- **Challenge 1.** The access to a data item in the remote node using one-sided RDMA verbs must know its memory address in the node a priori. However, in many cases, e.g., to fetch a proper version under MVCC and its variants, the memory address of a proper version to be read/written in the remote node cannot be known in advance without traversing the version chain for a given key. *To address this challenge, we make a specially designed key-to-address index in the framework, with which, for a given key, we can always obtain the memory address of the data in the remote node using up to 3 RDMA one-sided calls.* This access method builds a foundation to re-implement concurrency control algorithms using one-sided RDMA verbs only.
- **Challenge 2:** The facilitation and generalization of re-implementing mainstream concurrency control algorithms are necessary for a fair comparison. One-sided RDMA verbs, which are READ, WRITE, WRITE With Immediate, FETCH-And-ADD (a.k.a FAA), and COMPARE-And-SWAP (a.k.a CAS), cannot directly support locking/unlocking/validating operations of concurrency control algorithms. To address this challenge, we first collect and build metadata in the framework for the mainstream concurrency control algorithms. We then abstract six primitives that are used to manipulate the above metadata using the key-to-address index in the framework on top of the one-sided RDMA verbs [26]. The six primitives help do locking/unlocking/validating operations, which make us enjoy all benefits of RDMA networks and facilitate the re-implementation of the concurrency control algorithms. Note that, our proposed primitives are different from those designed in FaRM [18] which aim to speed up the message communication, while ours are used to read/write/modify the metadata or data items on remote nodes.

After building the framework RCBench, we then re-implement multiple mainstream state-of-the-art concurrency control algorithms, including (1) traditional protocols such as 2PL [23]: No-Wait [6], Wait-Die [37], Wound-Wait [37], and timestamp-based protocols: T/O [37, 5, 6], MVCC [49]; (2) advanced protocols such as Silo [44], Maat [24] and Cicada [31]; (3) a classic deterministic protocol Calvin [42]. Besides, we adopt various optimizations in the framework, including coroutine, doorbell batching, outstanding requests, and passive ack for the above algorithms.

We conduct comprehensive experiments over the re-implemented concurrency control algorithms over the commonly used benchmarks, and some important findings are reported in the following:

- It is convenient to re-implement the mainstream concurrency control algorithms using our proposed six primitives under RCBench. The results show that our re-implementations can even achieve a better performance than some customized implementations under the same settings.
- Optimizing Calvin using RDMA cannot bring obvious benefits, while other algorithms exhibit significant performance improvement (18X to 42X) using RDMA primitives. The degree of improvement closely relies on the number of primitive calls. In our

experiments, Silo is reported to perform the best in most cases.
– Among all optimization techniques together with RDMA, coroutine brings the greatest performance improvement (1.7X to 2.5X). Besides, the selection of proper lock types for 2PL is also important to affect the performance. For example, using a single type of exclusive locks instead of exclusive/shared locks can significantly improve performance in moderate or low contention scenarios.
– $\mathcal{U}^e_{CPU}$ is the dominant factor to scale out distributed transaction processing. By optimizing $\mathcal{U}^e_{CPU}$, transaction scalability can be arguably achieved. We must emphasize this conclusion is rather important, which helps eliminate the expensive data migration overhead. This is because transaction scalability can make a distributed transaction processing achieve a similar performance when the number of accessed data nodes per transaction increases.

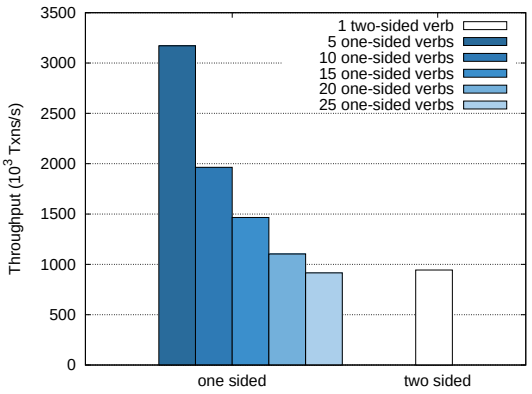


Fig. 3 A performance comparison between RDMA verbs

2 Background

2.1 Distributed Transaction Processing

In traditional shared-nothing distributed database systems, data are horizontally partitioned into ranges, and the data nodes are responsible for storing and accessing the ranges that are assigned to them. These systems adopt multi-coordinator system architecture to do distributed transaction processing, such as Percolator [34]. 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 termination of the transaction. In step 2), each data node is equipped with a database instance that manages the concurrent execution of the sub-transactions executing at this node. In step 3), once the consensus of a total order of sub-transactions in every data node is made, the coordinator coordinates the commit of the transaction; otherwise, it coordinates the abort of the transaction.

As shown in Figure 2, the transaction scalability of traditional distributed transaction processing suffers from low $\mathcal{U}^e_{CPU}$. The reason is two-fold. On the one hand, TCP/IP networks require message packing and unpacking from both senders and receivers. In the packing/unpacking process, multi-layer data encapsulation/decapsulation and the context switching between the application mode and kernel mode often consume many more CPU cycles to execute operations of transactions. On the other hand, the high-latency networks together with the asynchronous scheduling of sub-transactions significantly raise the probability of read/write conflicts, and lead to an increasing number of aborted transactions. As a result, increasing ineffective work to execute aborted transactions further deteriorates $\mathcal{U}^e_{CPU}$.

2.2 RDMA

RDMA is a conceptual extension of the direct memory access (DMA) technology, and is capable of directly accessing memory on a remote machine without the intervention of the remote CPU. It can be supported on both Ethernet and InfiniBand networks that provide the same common user APIs for programming. Compared with the Ethernet network, the InfiniBand network provides much higher bandwidth and lower latency. Hence, in this paper, we focus on RDMA-based programming over the InfiniBand network. Thus far, RDMA-based programming 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 time in the remote machine.

RDMA 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 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 the CPU involvement of the remote machine. Therefore, a one-sided verb always seems more favorable than a two-sided verb.

Although one-sided verbs enjoy 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. On the contrary, we can do the remote read/write by issuing a single two-sided verb. To roughly quantify how many one-sided verb invocations are performance equivalent to a single two-sided verb invocation, we report the experimental evaluation in Figure 3. In real experiment, the size of each data item to be read in every round-trip is set to 1KB, which is a common read size in our implementation. As can be seen, it brings benefits by issuing 20 or less than 20 one-sided verb invocations against a single two-sided verb invocation. Thus, in this paper, we attempt to use one-sided verbs to re-implement distributed transaction processing.

3 Overview of RCBench

Figure 4 presents an overview of RCBench to process distributed transactions using RDMA. The implementation of RCBench has already been publicly available on GitHub.[1] RCBench is based on shared-memory[8] distributed system architecture, in which nodes are interconnected via one-sided verbs. In RCBench, each node acts as both compute node and data node, and is equipped with 1) multiple **transaction executors** to execute transactions with serializability guarantees, and 2) a **MemStore** that is a buffer used to manage data and metadata in memory, and facilitate transaction executors to do concurrency control. Note that we do not consider fault tolerance, which is orthogonal to our work. Recent works on fault tolerance can be referred to [54, 47].

Every transaction 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. In our design, the transaction executor can work in two modes: thread-to-transaction and coroutine-to-transaction. In the first mode, a thread is created for one executor to execute transactions one by one; in the second mode, a thread is composed of multiple coroutines, each of which is created for one executor to execute transactions one by one. 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, and consequently improves the effective CPU utilization rate.

[1] https://github.com/rhaaaa123/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
$T^l.bts$	The beginning timestamp of T
$T^l.cts$	The commit timestamp of T
$T^g.lock$	The lock used to prevent concurrent writes on T^g
$T^g.st$	The $running/aborted/committed$ status of T
$T^g.lb$	The lower bound of T's timestamp interval
$T^g.ub$	The upper bound of T's timestamp interval
$X^m.lock$	The lock used to prevent concurrent writes on X^d and X^m
$X^m.st$	The status of the transaction with the latest write on X
$X^m.pL$	A list of pairs $\langle\ T^l.bts, T.Tid\ \rangle$ where T is granted with a lock on X
$X^m.rts$	The maximum beginning/commit timestamp of transactions that have ever read X
$X^m.wts$	The maximum beginning/commit timestamp of transactions that have ever written X
$X^m.wid$	The ID of the uncommitted transaction that is granted with an exclusive lock on X
$X^m.rL$	An ID list of running transactions that have ever read X
$X^m.wL$	An ID list of running transactions that have ever written X
$T^l.grts$	$\max\{X^m.rts\}$ for $\forall X \in T^l.ws$
$T^l.gwts$	$\max\{X^m.wts\}$ for $\forall X \in T^l.ws \cup T^l.rs$
$T^l.bL$	A list of $T'.Tid$ where T' is ordered before T
$T^l.aL$	A list of $T'.Tid$ where T' is ordered after T
$T^l.oL$	A list of $T'.Tid$ where T' is not in $T^l.aL$ and $T^l.bL$, but T' modifies the same data items with T

To guarantee serializability, we design two categories of metadata in each node used for concurrency control. The first category is called ItemMeta X^m associated with data X^d in each data item X. X^m can be a read/write lock identifier (e.g., the ID of a transaction that has ever read/written X) in 2PL or the most recent timestamp of transactions that have ever read/written X in T/O, or others. X^d and X^m of each data item X are continuously stored in MemStore, and

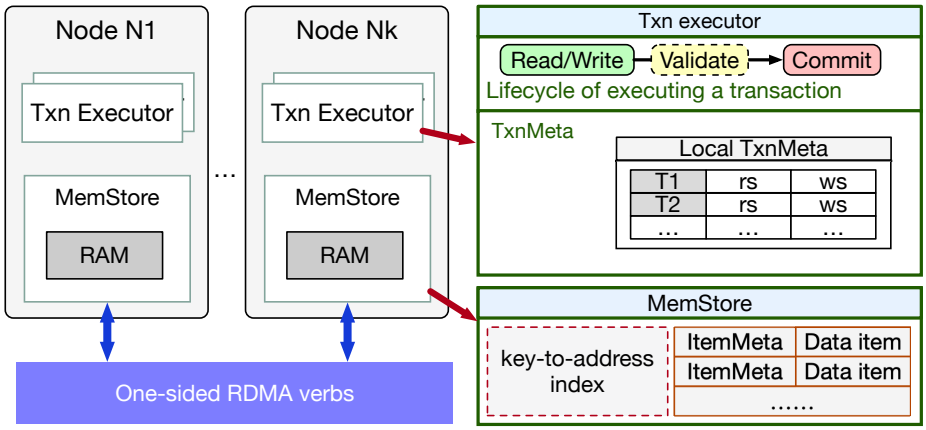


Fig. 4 An overview of RCBench

can be read/written/updated by transaction executors from the remote nodes. The second category is called TxnMeta, which is created for each transaction. TxnMeta is further divided into local TxnMeta and global TxnMeta. Local TxnMeta T^l can be a transaction's read/write set, begin/commit timestamp, etc. T^l can only be read/written/updated by the local transaction executors. Global TxnMeta T^g can be a transaction's running/aborted/committed status, or/and a timestamp interval used for ordering the transactions. T^g can be read/written/updated by transaction executors from the remote nodes. For ease of illustration, we list symbols and their meanings in Table 1 that will be used throughout this paper. Together with these metadata, we propose six primitives in RCBench to re-implement the mainstream concurrency control algorithms, and elaborate on them in the next section.

4 Access to Remote Metadata

By making a thorough investigation of mainstream concurrency control algorithms, we conclude that these algorithms can be implemented completely based on the manipulation of metadata (i.e., X^m, T^g, and T^l). Thus, in this section, we first elaborate on what the metadata is in each concurrency control algorithm, and then abstract six primitives to manipulate metadata X^m and T^g that are located in the remote data nodes using one-sided verbs. With these six primitives, we are able to make the re-implementations (shown in Section 5) of the state-of-the-art algorithms that are transparent to RDMA-based programming and capable of enjoying all benefits of RDMA networks.

4.1 Metadata

We investigate four popular categories of concurrency control algorithms and summarize the used metadata individually in Figure 5.

• **Lock-based algorithms**. No-Wait, Wait-Die, and Wound-Wait are three widely used variants for deadlock-free 2PL algorithms. For No-Wait, X^m includes lock meta $X^m.lock$, which occupies 8 bytes, with the lowest bit indicating the lock type $lock\_type$ (0: shared lock SL; 1: exclusive lock EL), and the remaining 63 bits indicating the number num_of_locks of shared locks on X. Note that an exclusive lock on X is granted (i.e. setting $X^m.lock.lock\_type = 1$) only if $X^m.lock.num\_of\_locks = 0$. T^l in No-Wait includes read set $T^l.rs$ and write set $T^l.ws$ of transaction T. For Wait-Die, X^m includes two fields: (1) another kind of lock meta $X^m.lock\_B$, which occupies 8 bytes as well to record the beginning timestamp of the transaction with an exclusive lock on X; and (2) a 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 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. In this way, if T cannot find an empty slot in $X^m.pL$, to make a correct schedule of executing transactions, we let T abort. Besides $T^l.rs$ and $T^l.ws$, T^l in Wait-Die includes the beginning timestamp ($T^l.bts$) of transaction T. For Wound-Wait, X^m includes two fields: (1) lock meta $X^m.lock$, and (2) the list $X^m.pL$. T^l maintains the same metadata as that in Wait-Die, while T^g additionally maintains a transaction status $T^g.st$, which is $running$ (RN), $committed$ (CM) or $aborted$ (AB).

• **Timestamp-based algorithms**. T/O and MVCC are two widely used methods for timestamp-based algo-

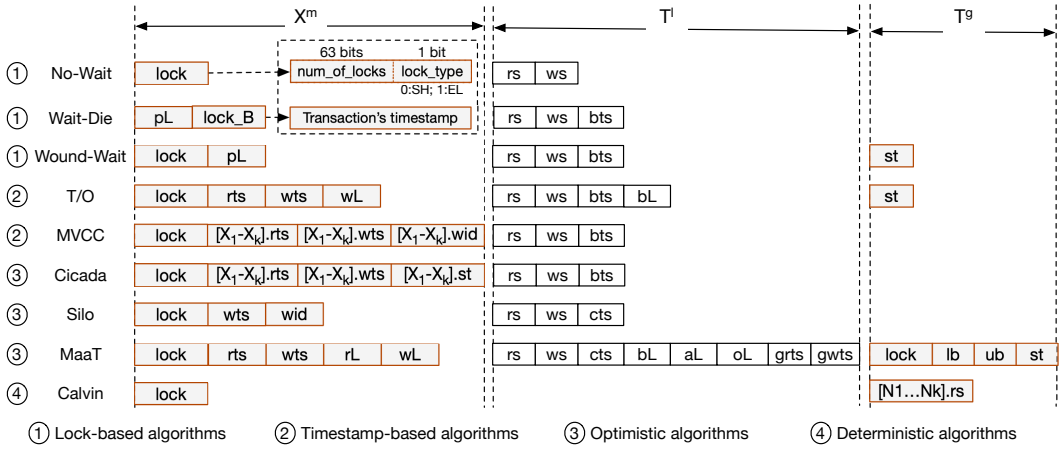


Fig. 5 Data structures of metadata used for concurrency control algorithms.

rithms. For T/O, X^m includes four fields: (1) $X^m.lock$, the same field used in No-Wait but acts as a latch, (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 has ever written X. Besides $T^l.rs$, $T^l.ws$ and $T^l.bts$, to support cascading aborts, T^l maintains an extra list $T^l.bL$, recording transactions that are ordered before T. For example, when a new transaction T writes a data item X, transactions that have ever written X should be ordered before T. To do this, we copy transactions in $X^m.wL$ and put them into $T^l.bL$ so that T is guaranteed to commit after transactions in $T^l.bL$ commit; otherwise, if any transaction in $T^l.bL$ aborts, we will let T do a cascading rollback. T^g maintains the same metadata as that in Wound-Wait. For MVCC, X^m includes lock meta $X^m.lock$, which acts as a latch. Each data item keeps a fixed number (e.g. 4) of version slots, each of which stores a version X_i. By doing this, we can read back all versions of that data item by issuing an RDMA READ. For each version X_i (considered as an individual data item) in MVCC, X_i^m includes 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 maintains the same metadata as that in Wait-Die.

• **Optimistic algorithms**. Silo, MaaT, and Cicada are three state-of-the-art optimistic algorithms. For Silo, X^m includes three fields: (1) lock meta $X^m.lock$, (2) $X^m.wts$, the maximum commit timestamp of transactions that have ever written X, and (3) $X^m.wid$, the ID of a transaction that is currently granted with an exclusive lock on X. Besides $T^l.rs$ and $T^l.ws$, T^l includes the commit timestamp ($T^l.cts$) of transactions T. For MaaT, X^m includes five fields: (1) lock meta $X^m.lock$, which acts as a 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. 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 lock meta $T^g.lock$, 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. For Cicada, the metadata are 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.

• **Deterministic algorithms**. Calvin is a classic deterministic concurrency control algorithm. For Calvin, X^m only includes lock meta $X^m.lock$. For each sub-transaction Ts, Ts^g includes read sets from other data nodes Ni ($Ts^g.Ni.rs$).

Note that, in T/O, MaaT, MVCC, and Cicada, lock meta $X^m.lock$ is used as a latch to prevent concurrent modifications on X.

4.2 Concurrency control primitives

To facilitate the manipulation of metadata, we abstract six primitives, with three primitives to access remote X and the other three primitives to access remote T^g.

4.2.1 Access Remote X

Let $\hat{X}$ be a triple of $\langle X.addr, X.size, X.PK\rangle$ where $X.addr$, $X.size$ and $X.PK$ are the memory address, size and primary key of X, respectively. In our design, each $\hat{X}$ occupies 24 bytes, with the first 8 bytes to store $X.addr$, the middle 8 bytes to store $X.size$, and the last 8 bytes to store $X.PK$ (we assume $X.PK$ is an integer/float/double). Given a primary key PK, we now illustrate how to obtain $\hat{X}$ with $\hat{X}.X.PK = PK$ using one-sided verbs.

We allocate a large, contiguous buffer in each MemStore, to build an RDMA-friendly key-to-address hash index IDX, where each key is a primary key $X.PK$, and its value is $\hat{X}$. Following the empirical methods in [32], we design IDX as a 3-way cuckoo hash table, i.e. three orthogonal hash functions are used in IDX. Specifically, given a primary key PK, three potential positions calculated from three hash functions are tried sequentially for its insertion in IDX. In case all are filled, one of them will be preempted by kicking the original PK' at that position and replacing it with PK. Then PK' tries to find its new position following the same procedure, which may result in successive kicks. When the number of kicks reaches a pre-defined limit or when a cycle is detected, the hash table will be resized for larger accommodation. As a result, it is guaranteed that any $\hat{X}$ with $\hat{X}.X.PK = PK$ is located in one of the three positions defined in Equation 1. That is, we can find the $\hat{X}$ with $\hat{X}.X.PK = PK$ using at most three seeks in IDX; otherwise, such $\hat{X}$ does not exist.

$$\hat{X}_i.addr = IDX + 24 \times hash_i(PK),\ \text{i=1,2,3} \quad (1)$$

We design function *getRtItemAddr* to obtain the remote $\hat{X}$ for a given PK using the following steps: (1) locally calculate one potential position $\hat{X}_i.addr$ according to Equation 1, (2) issue an RDMA READ to fetch remote $\hat{X}$ based on $\hat{X}_i.addr$, and (3) return $\hat{X}_i$ if $\hat{X}_i.X.PK = PK$. In step (3), if $\hat{X}_i.X.PK \neq PK$, we redo (1)(2)(3) until we either find a correct $\hat{X}_i$ with $\hat{X}_i.X.PK = PK$, or retry with extra two failed attempts, which imply the non-existence of requested $\hat{X}$. Remind that in Section 2.2, the cost of 20 RDMA one-sided calls is almost equivalent to that of one two-sided verb call. As [32] reports an average of 1.6 and a maximum of 3 one-sided verb calls to obtain the requested $\hat{X}$ for a given PK, the design of IDX makes our one-sided RDMA programming significantly faster than two-sided RDMA programming.

It is worth mentioning that, in practice, we do not necessarily make our assumption that the primary key is an integer/float/double. That is, if $X.PK$ is of variable length such as a string, we do not maintain $X.PK$ in $\hat{X}$. In this way, to verify whether we find a correct $\hat{X}_i$, we need to issue another RDMA READ to read $X.PK$.

After obtaining $\hat{X}$, we now propose three primitives that are **ReadDI, WriteDI, AModIM** to manipulate remote X, including both X^d and X^m.

Primitive 1: ReadDI(PK)

```
1 X̂←getRtItemAddr(PK);
2 if X̂ = NULL then return NULL;
3 return RDMA_READ(X̂.X.addr, X̂.X.size)
```

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

Primitive 2: WriteDI($PK, newV$)

```
1 X̂←getRtItemAddr(PK);
2 if X̂ = NULL then return false;
3 RDMA_WRITE(X̂.X.addr,newV,X̂.X.size);
4 return true;
```

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

Primitive 3: AModIM($PK, meta, oldV, newV$)

```
1 X̂←getRtItemAddr(PK);
2 if X̂ = NULL then return false;
3 X.meta_addr ← X̂.X.addr + meta.offset;
4 t←RDMA_CAS(X.meta_addr, oldV, newV);
5 return t = oldV;
```

– **AModIM** (Primitive 3) is designed to conditionally update an item ($meta$) of X^m, e.g. $X^m.lock$ or $X^m.rts$, with the atomicity guarantee. Similar to ReadDI, AModIM issues $getRtItemAddr$ to obtain $\hat{X}$ (line 1). If $\hat{X}$ exists, AModIM calculates the remote address $X.meta\_addr$ of $meta$ by adding $meta$'s offset in X^m to $X.addr$ (line 3), and issues RDMA CAS to conditionally update $meta$ by new value $newV$ with atomicity guarantee (line 4).

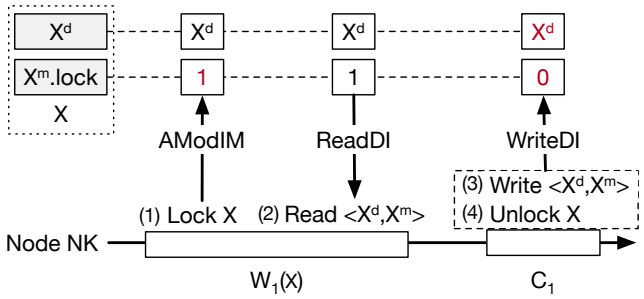


Fig. 6 Using primitives to access remote X in No-Wait.

For illustration purposes, we give an example to show how the primitives are used in Figure 6. In this example, a transaction T_1 modifies a remote data item X (denoted as $W_1(X)$) and commits (denoted as C_1). We assume No-Wait algorithm is used to do concurrency control. To modify X with $W_1(X)$, T_1 sequentially (1) acquires an exclusive lock on X, and (2) reads X to the local node; then, we generate a local copy X' of X and apply writes to X'. To commit with C_1, T_1 (3) writes X' to replace the original X, and (4) releases the lock. For step (1), T_1 issues an AModIM to acquire the lock on X by modifying $X^m.lock$ from 0 to 1 (highlighted in red) atomically. For step (2), T_1 issues a ReadDI to read X. Finally, for steps (3) and (4), T_1 re-sets $X'^m.lock$ to 0 (highlighted in red) locally, then issues a WriteDI to overwrite the original X by X'.

4.2.2 Access Remote T^g

Similarly, we allocate another large, contiguous buffer buf_{T^g} in MemStore that maintains T^g for each transaction T. In our design, T^g occupies a fixed-length Θ of space, and $T.Tid$ is taken as an integer. Thus, we implement the buffer as a list, where T^g is put in a proper offset (defined in Equation 2) of the buffer. Note that Δ helps set T^g in a round-robin fashion. By doing this, we design function $getRtTgAddr$ to obtain the remote T^g for a given Tid using the following steps: (1) locally calculate $T^g.addr$ in MemStore of the remote data node according to Equation 2, (2) issue a one-sided verb to get T^g, and (3) return T^g if $T.Tid = Tid$. Otherwise, it means that the requested T^g no longer exists.

$$T^g.addr = buf_{T^g} + \Theta \times Tid\%\Delta \quad (2)$$

After obtaining $T^g.addr$, we propose another three primitives that are **ReadTM, WriteTM, AModTM**, to manipulate remote T^g.

Primitive 4: ReadTM(Tid)

```
1 T^g.addr ← getRtTgAddr(Tid) //Equation 2;
2 ⟨ T.Tid, T^g⟩ ← RDMA_READ(T^g.addr, Θ) ;
3 if T.Tid = Tid then return T^g;
4 else return NULL;
```

– **ReadTM** (Primitive 4) is designed to read remote T^g. In each node Ni, we maintain the address of buf_{T^g} of every data node. By giving Tid as input, ReadTM calculates the remote address $T^g.addr$ using $getRtTgAddr$ function (line 1) and issues an RDMA READ to read T^g via $T^g.addr$ and its fixed length Θ (line 2). Finally, T^g is returned if it appears to be the requested one with the given Tid; otherwise, $NULL$ is returned as a sign of non-existence (lines 3,4).

Primitive 5: WriteTM($Tid, newV$)

```
1 T^g.addr ← getRtTgAddr(Tid) //Equation 2;
2 ⟨ T.Tid, T^g⟩ ← RDMA_READ(T^g.addr, Θ) ;
3 if T.Tid ≠ Tid then return false;
4 RDMA_WRITE(T^g.addr, newV, Θ);
5 return true;
```

– **WriteTM** (Primitive 5) is designed to write remote T^g. Similar to ReadTM, WriteTM calculates the remote address $T^g.addr$ locally (line 1), and issues an RDMA READ to examine whether T^g in the given address has a matching Tid (line 2). If the examination fails, WriteTM returns $false$ (line 3); otherwise, it issues an RDMA WRITE to overwrite T^g with $newV$ of fixed length Θ (line 4).

Primitive 6: AModTM($Tid, meta, oldV, newV$)

```
1 T^g.addr ← getRtTgAddr(Tid) //Equation 2;
2 ⟨ T.Tid, T^g⟩ ← RDMA_READ(T^g.addr, Θ) ;
3 if T.Tid ≠ Tid then return false;
4 T^g.meta_addr ← T^g.addr + meta.offset;
5 t←RDMA_CAS(T^g.meta_addr, oldV, newV);
6 return t = oldV;
```

– **AModTM** (Primitive 6) is designed to conditionally update an item (*meta*) of T^g, e.g., $T^g.lock$, $T^g.st$ with atomicity guarantee. Similar to ReadTM, AModTM first calculates $T^g.addr$ (line 1) and issues an RDMA READ to examine whether the T^g in the given address has a matching Tid (line 2,3). If the examination fails, AModTM returns $false$; otherwise, it calculates $T^g.meta\_addr$ of *meta* by adding *meta*'s offset in T^g to $T^g.addr$ locally (line 4), and then issues RDMA CAS to conditionally update *meta* by new value $newV$ with atomicity guarantee (line 5).

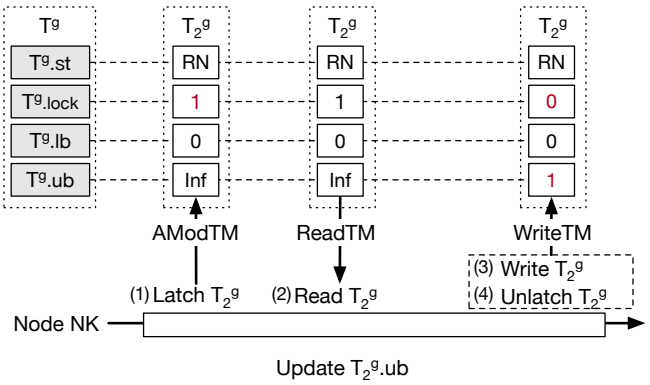


Fig. 7 Using primitives to access remote T^g in MaaT.

We give an example of timestamp interval adjustment in MaaT to show how the primitives are used in Figure 7. In this example, a transaction T_1 modifies T_2's timestamp interval [$T_2^g.lb$,$T_2^g.ub$] to guarantee that the remote transaction T_2 is ordered before T_1. To do this, T_1 sequentially (1) acquires a latch on T_2^g, (2) reads T_2^g, and locally calculates T_2's new timestamp interval, (3) writes the updated T_2^g, and finally (4) releases the latch. For step (1), T_1 issues an AModTM to acquire the lock on T_2^g by updating $T_2^g.lock$ from 0 to 1 (highlighted in red) atomically. For step (2), T_1 issues a ReadTM to read T_2^g, and locally sets the new value of $T_2^g.ub$ to 1. Finally, for step (3) and (4), T_1 re-sets $T_2^g.lock$ to 0 and sets $T_2^g.ub$ to 1 by issuing a WriteTM.

5 Re-implementations

In this section, we first present how to re-implement concurrency control algorithms completely based on the proposed six primitives, and then discuss the optimizations.

5.1 Lock-based Algorithms

No-Wait [6] 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 every read or write on X and aborts immediately in case of locking failures to avoid deadlocks. To boost No-Wait using RDMA, it is necessary to re-implement its logic that (1) acquire remote locks, (2) perform remote reads/writes, and (3) release remote locks. Key functions of the re-implementation called RDMA-No-Wait are shown in Algorithm 1.

Algorithm 1: RDMA-No-Wait

```
1  Function AcqIMShLock(PK):
2      X ← ReadDI(PK);
3      if X^m.lock .lock_type = EL then
4          return false;
5      else
6          newL ←X^m.lock;
7          newL.num_of_locks++;
8          r ← AModIM(PK, MT_LOCK, X^m.lock,
             newL);
9          return r;
10 Function AcqIMExcLock(PK):
11     r ← AModIM(PK, MT_LOCK, 0, EL);
12     return r;
13 Function RlsIMShLock(PK):
14     X ← ReadDI(PK);
15     newL ←X^m.lock;
16     newL.num_of_locks − −;
17     r ← AModIM(PK, MT_LOCK,X^m.lock
         , newL);
18     if ¬r then goto line 14;
19 Function RlsIMExcLock(PK):
20     AModIM(PK, MT_LOCK, EL, 0));
21 Function Read(PK, T):
22     if ¬AcqIMShLock(PK) then Abort T;
23     X ← ReadDI(PK);
24     T^l.rs ← {X}∪ T^l.rs;
25 Function Write(PK, newV, T):
26     if ¬ AcqIMExcLock(PK) then Abort T;
27     X ← ReadDI(PK); X^d ← newV;
28     T^l.ws ← {X}∪ T^l.ws;
29 Function Commit(T):
30     foreach X ∈ T^l.rs do
31         RlsIMShLock(X.PK );
32     foreach X ∈ T^l.ws do
33         X^m.lock ←0; WriteDI(X.PK , X);
```

Functions *AcqIMShLock* and *AcqIMExcLock* are used to acquire remote shared locks and exclusive locks, respectively, based on primitives *ReadDI* and *AModIM*. By taking its primary key PK as the input, to acquire a shared lock on X with $X.PK = PK$, we first issue a remote read of X using primitive *ReadDI* (line 2) and check whether there is an exclusive lock on X locally (line 3). 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 (line

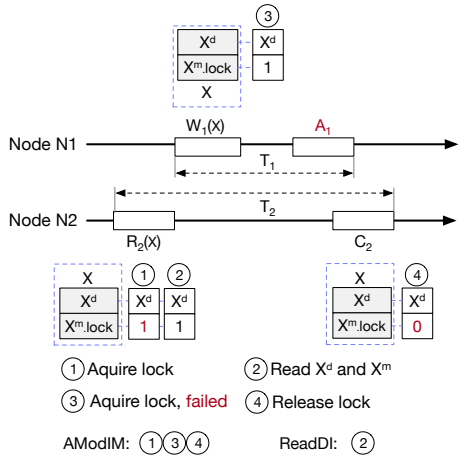


Fig. 8 An example of RDMA-No-Wait.

6–7); we then issue a remote write to update $X^m.lock$ with $newL$ to declare that a new shared lock on X is granted (line 8). Note that, parameter MT_LOCK used in Function *AModIM* refers to meta *lock* of ItemMeta; *AModIM* 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 has made between *ReadDI* (line 2) and *AModIM* (line 8), the remote write would fail, and the returned value r is set to be false (line 9). To acquire an exclusive lock on data item X, we directly employ primitive *AModIM* to update the lock meta of X^m. If there is either an exclusive lock or a shared lock on X, the update fails. Following the reverse logic of *AcqIMShLock* and *AcqIMExcLock*, functions *RlsIMShLock* and *RlsIMExcLock* are used to release remote shared locks and exclusive locks, respectively (line 13–20). We omit the details to avoid repeated expressions.

Functions *Read* and *Write* are used to do remote reads and writes, respectively, based on primitive *ReadDI*. For *Read*, we first try to acquire a shared lock on X with $X.PK = PK$. If the lock acquisition fails, we abort the transaction; otherwise, we read the remote X (this operation can be omitted because X has been fetched in line 2) and add it to the local read set $T^l.rs$ (line 22-24). Similarly, for *Write*, we try to acquire an exclusive lock on X. If we fail to acquire the lock, we abort the transaction; otherwise, we read the remote X, update X^d locally, and add X to the local write set $T^l.ws$ (line 26-28).

Upon commit or abort of transaction T, we release the locks that T has acquired. For ease of illustration, we show how to do commit in Function *commit*. We sequentially release the shared locks that T has acquired (line 30–31) using *RlsIMShLock*. For each data item, X in the write set $T^l.ws$, we first issue a remote write of X using primitive *WriteDI* and then release the exclusive lock using *RlsIMExcLock*. The latter can be integrated into primitive *WriteDI* by setting $x^m.lock$ to 0 (line 32–33) to reduce one primitive call.

Discussion. As shown in Algorithm 1, it requires at least two primitive calls (*ReadDI* and *AModIM*) to acquire/release a shared lock while it requires only one primitive call (*AModIM*) 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 the a 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 6.3, 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 1. For illustration purposes, we show how the single exclusive locking mechanism works in Example 1.

Example 1 We consider two concurrent transactions T_1 and T_2 in Figure 8. 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 AModIM call (line 22, step ①); T_2 then issues a ReadDI 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 AModIM 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 [37] 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,

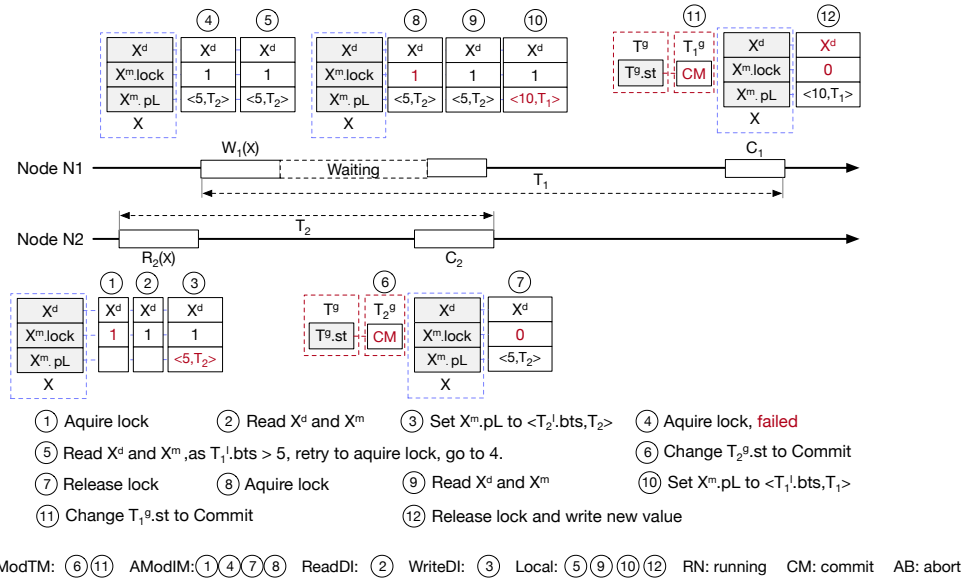


Fig. 9 An example of RDMA-Wound-Wait.

Algorithm 2: RDMA-Wound-Wait

```
1  Function LockWithWound(PK, T):
2      while ¬ AcqIMExcLock(PK) do
3          X ← ReadDI(PK);
4          ⟨ T̄^l.bts , T̄.Tid ⟩ ← X^m.pL;
5          if T^l.bts < T̄^l.bts then
6              AModTM(T̄.Tid, MT_ST, RN, AB);
7      X ← ReadDI(PK);
8      X^m.pL ← ⟨ T^l.bts , T.Tid ⟩;
9      WriteDI(PK, X);
10     return X;
11 Function Commit(T):
12     if ¬AModTM(T̄.Tid, MT_ST, RN, CM) then
           Abort T;
13     foreach X ∈ T^l.ws do
14         X^m.lock ←0; WriteDI(X.PK , X);
15     foreach X ∈ T^l.rs do
16         RlsIMExcLock(X.PK);
```

it also requires to (1) acquire remote locks, (2) do remote reads/writes, and (3) release remote locks. Key functions of the re-implementation called RDMA-Wound-Wait are shown in Algorithm 2.

Function *LockWithWound* is used to acquire remote locks, based on *AModIM*, *ReadDI* and *WriteDI* primitives. First, we issue *AcqIMExclock* (given in Algorithm 1) 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 *ReadDI*, set the lock owner on X to T, and perform a remote write of X using *WriteDI* 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 *ReadDI* (line 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 doing a remote write of 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 Example 2, we present how RDMA-Wound-Wait works.

Example 2 Consider Figure 8. Suppose $T_1^l.bts = 10$ and $T_2^l.bts = 5$. We show every step of executing RDMA-Wound-Wait in Figure 9. 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 [37] 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 *AModTM* by *Return Abort* (line 6) in algorithm 2.

5.2 Timestamp-based Algorithms

T/O orders transactions based on their beginning timestamps. 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. Key functions of the re-implementation called RDMA-T/O are shown in Algorithm 3.

Function *Write* is used to do remote write based on *AModIM*, *ReadDI* and *WriteDI*. We first try to acquire a remote latch on X to prevent concurrent modifications by issuing *AModIM* (line 2). If the latch is granted, we issue *ReadDI* to do a remote read of X and check whether X is writable by comparing $\max\{X^m.rts, X^m.wts\}$ with $T^l.bts$ (line 3–4). 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 (line 4); 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 5). Subsequently, we do a local update on $X^m.wts$, X^d, $T^l.bL$ (the dependent transactions of T), $X^m.wL$ (the running transactions with writes on X), $X^m.lock$ (the latch on X), and finally we apply the local updates on remote X by issuing *WriteDI* (line 6–9).

Function *Read* is used to do remote read based on *ReadDI* and *AModIM*. We first read X by issuing *ReadDI* (line 15), and check whether X is readable by T (line 16). If X is not readable by T, we abort T; otherwise, we try to do a remote update on $X^m.rts$ using $T^l.bts$ and

Algorithm 3: RDMA-T/O

```
1  Function Write(PK, T, newV):
2      if ¬ AcqIMExcLock(PK) then Abort T;
3      X ←ReadDI(PK);
4      if max{X^m.rts,X^m.wts } > T^l.bts then
         RlsIMExcLock(PK); Abort T ;
5      T^l.ws ←{X}∪ T^l.ws;
6      X^m.wts ←T^l.bts; X^d ← newV;
7      T^l.bL ← X^m.wL ∪ T^l.bL;
8      X^m.wL ← X^m.wL ∪ {T.Tid };
9      X^m.lock ← 0; WriteDI(PK, X);
10 Function UpdateRTS(PK, X, T):
11     if X^m.rts < T^l.bts then
12         AModIM(PK, MT_RTS,X^m.rts ,T^l.bts );
             X^m.rts ←T^l.bts;
13         if X ≠ ReadDI(PK) then Abort T;
14 Function Read(PK, T):
15     X ←ReadDI(PK);
16     if X^m.wts > T^l.bts or X^m.lock ≠ 0 then
17         Abort T;
18     UpdateRTS(X, PK);
19     T^l.bL ← X^m.wL ∪ T^l.bL;
20     T^l.rs ← {X}∪ T^l.rs; return X;
21 Function CascadingAbortCheck(T):
22     foreach T̄id ∈ T^l.bL do
23         do T̄ ← ReadTM(T̄id) ;
24         while T̄^g.st = RN;
25         if T̄^g.st = AB then
26             T^g.st ← AB; return false;
27     T^g.st ← CM; return true;
28 Function Commit(T):
29     if ¬CascadingAbortCheck(T) then Abort T ;
30     foreach X ∈ T^l.ws do
31         do res← AcqIMExcLock(X.PK) ;
32         while ¬res;
33         X̄ ← ReadDI(X.PK);
34         X̄^m.wL ← X̄^m.wL - T.Tid;
35         X̄^m.lock ← 0; WriteDI(PK, X̄);
```

[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 $T^l.bts < X^m.wts$.

perform another remote read on X to check whether the remote update is successful (lines 11–13). In our case, we use double-read instead of acquiring a latch on X to guarantee atomicity. This is because, by doing this, we can eliminate the lock holding time on X to improve the concurrency. We abort T if the update fails; otherwise, we perform a local update on $T^l.bL$ and store X to the read set $T^l.rs$ (lines 19–20).

Upon commit of transaction T, it is necessary to check the status of each dependent transaction, maintained in $T^l.bL$, of T (lines 22–26). If any of them aborts, we would make a cascading abort of T (lines 25–26); if all of them commit, we set the status $T^g.st$ of T to CM 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 (line 30–35). 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. For illustration purposes, we present how RDMA-T/O works in Example 3.

Example 3 Consider Figure 8. We show every step of executing RDMA-T/O in Figure 10. For $R_2(X)$, T_2 does a remote read on X, sets $X^m.rts$ to $T_2^l.bts$ by issuing AModIM, 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). □

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

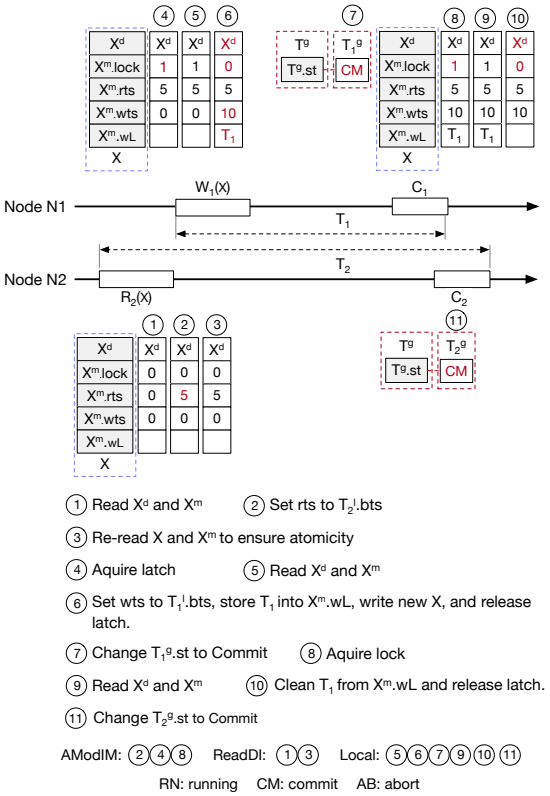


Fig. 10 An example of RDMA-T/O.

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 [6] to adjust RDMA-T/O without cascading abort, and report the experimental evaluation over the two implementations in Section 6.3.

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 [4] 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

Algorithm 4: RDMA-MVCC

```
1  Function UpdateVersionRTS(PK, X, T):
2      if X_i^m.rts < T^l.bts then
3          AModIM(PK, MT_X_i_RTS, X_i^m.rts
             , T^l.bts );
4          X_i^m.rts ← T^l.bts;
5          if X ≠ ReadDI(PK) then Abort T;
6  Function Read(PK, T):
7      X ← ReadDI(PK); // X includes its all versions;
8      X_i ← getProperVer(X, T);
9      if X_i=NULL or X_k^m.wid ≠0 or X^m.lock ≠0
         then Abort T;
10     UpdateVersionRTS(PK, X, T);
11 Function Write(PK, T, newV):
12     if ¬ AcqIMExcLock(PK) then Abort T;
13     X ← ReadDI(PK);
14     X_k ← getLatestVer(X);
15     if max{X_k^m.rts, X_k^m.wts } > T^l.bts or X_k^m.wid
         ≠ 0 then
16         RlsIMExcLock(PK); Abort T;
17     create a new version X_{l+1} of X by setting X_{k+1}^d
         to newV ;
18     X_{k+1}^m.wts ← T^l.bts; X_{k+1}^m.wid ← T.Tid;
19     X_k^m.wid ← T.Tid; X^m.lock ← 0;
20     T^l.ws ← {X}∪ T^l.ws;
21     WriteDI(PK, X);
22 Function Commit(X, T, newV):
23     foreach X ∈ T^l.ws do
24         AModIM(X.PK, MT_X_k_WID, T.Tid, 0);
25         AModIM(X.PK, MT_X_{k+1}_WID, T.Tid,
             0);
```

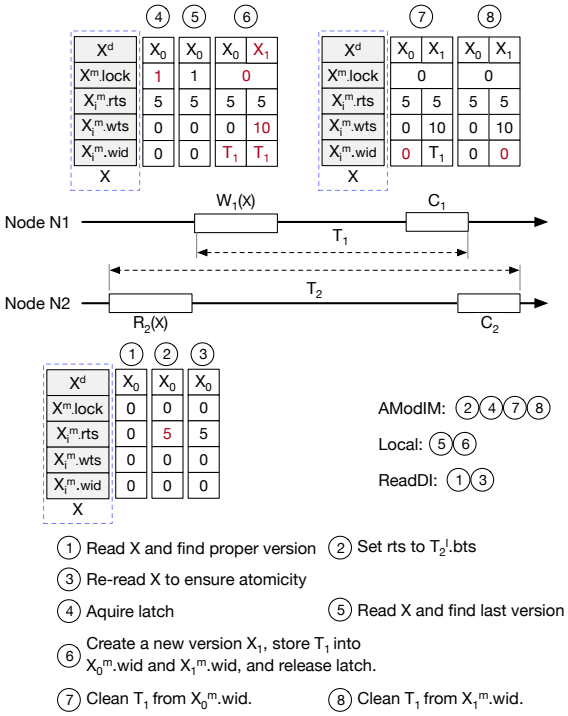


Fig. 11 An example of RDMA-MVCC.

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; 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 $\max\{X_{k+1}^m.rts, T^l.bts\}$. Note that, each version X_i of X separately maintains meta $X_i^m.rts$, $X_i^m.wts$ and $X_i^m.wid$, but all versions share the lock meta $X^m.lock$. 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 $X_i^m.wts$. Key functions of the re-implementation called RDMA-MVCC are shown in Algorithm 4. Function *Read* is used to do remote read based on primitives *ReadDI* and *AModIM*. We first perform a remote read to fetch X with all its versions by issuing *ReadDI*. 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 *AModIM*, *ReadDI* and *WriteDI*. 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 15–16). 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 *WriteDI* (line 17–21). 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 17).

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 *AModIM* (line 24,25), in

Algorithm 5: RDMA-Silo

```
1  Function Read(PK, T):
2      X ← ReadDI(PK); T^l.rs ← {X} ∪ T^l.rs;
3  Function Write(PK, T, newV):
4      X ← ReadDI(PK); X^d ← newV;
5      T^l.ws ← {X} ∪ T^l.ws;
6  Function Validation(T):
7      T^l.cts ← get the current timestamp;
8      foreach X ∈ T^l.ws do
9          if ¬ AcqIMExcLock(X.PK) then
10             Abort T;
11         AModIM(X.PK, MT_WID, 0, T.Tid);
12     foreach X ∈ T^l.rs ∪ T^l.ws do
13         X̄ ← ReadDI(X.PK);
14         if X̄^m.lock ≠ 0 and X̄^m.wid ≠ T.Tid then
               Abort T;
15         if X̄^m.wts ≠ X^m.wts then Abort T;
16 Function Commit(T):
17     foreach X ∈ T^l.ws do
18         X^m.wts ← T^l.cts; X^m.wid ← 0;
19         X^m.lock ←0; WriteDI(X.PK, X);
```

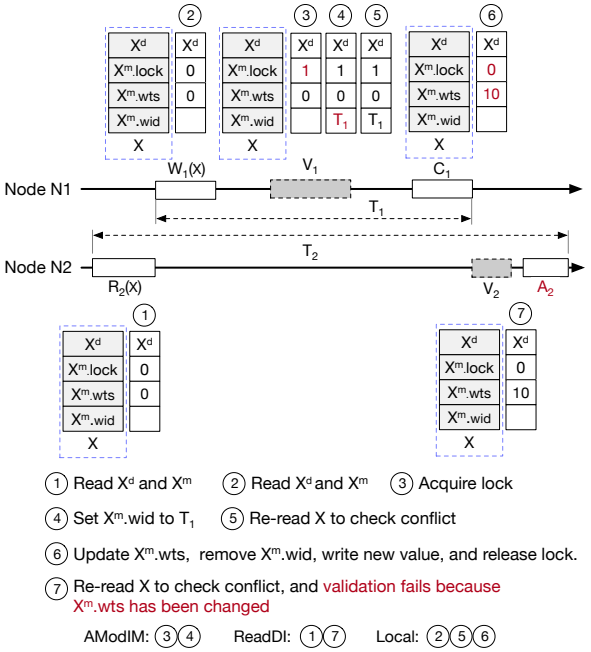


Fig. 12 An example of RDMA-Silo.

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 Example 4.

Example 4 Consider Figure 8. We show every step of executing RDMA-MVCC in Figure 11. 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 AModIM, 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 AModIM (step ⑦ – ⑧). □

5.3 Optimistic Algorithms

Silo [44] 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 is necessary 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. Key functions of the re-implementation called RDMA-Silo are shown in Algorithm 5.

Functions *Read* and *Write* are used to perform the remote read/write on data item X based on primitives *ReadDI*. For *Read*, we read remote X, and add it to local read set $T^l.rs$ (line 2). For *Write*, we read remote X, update X^d locally and add X to local write set $T^l.ws$ (line 4–5). Function *Validation* is used to perform 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. If the lock acquisition fails, T aborts; otherwise, we set T as the lock owner of X by issuing *AModIM* (line 8–11). Subsequently, $\forall X \in T^l.ws \cup T^l.rs$, we examine whether X has been modified by other transactions based on a double read of X using *ReadDI* (line 12–13, note that the first read was executed in line 2 or line 4). If X has been locked or modified, we abort T (lines 14–15). Upon commit of T, $\forall X \in T^l.ws$, we make a single remote write by issu-

ing *WriteDI* to release the lock on X, as well as update $X^m.wts$ and 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 Example 5.

Example 5 Consider Figure 8. We present values of metadata including X^d, $X^m.lock$, $X^m.wts$ and $X^m.wid$ at every step of executing RDMA-Silo in Figure 12. For $R_2(X)$, T_2 does a remote read on X, and stores X in $T_2^l.rs$ (step ①). For $W_1(X)$, T_1 does a local read on X, updates X^d locally, and stores X in $T_1^l.ws$ (step ②). In the validation phase of T_1, first, T_1 acquires the lock on X, and updates $X^m.wid$ by issuing two *AModIM* calls. Then, T_1 does a local read on X to ensure that X has not been modified by other transactions (step ③ – ⑤). Next, T_1 commits. T_1 updates X^d, $X^m.wts$, removes T_1 from $X^m.wid$, and releases the lock (step ⑥). 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 ⑦). □

Discussion. Similar to other optimistic concurrency control algorithms, Silo makes an implicit assumption about the application scenarios with the low-contention workload. To handle the high-contention workload, MOCC [46] introduces a pessimistic locking mechanism to the optimistic concurrency control algorithm to do hybrid concurrency control. Due to the space limitation, we omit the details of re-implementing MOCC (called RDMA-MOCC) using RDMA but make a thorough experimental evaluation between RDMA-Silo and RDMA-MOCC in Section 6.3.

MaaT [24] 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 $[T^g.lb, T^g.ub]$ [9] 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. Key functions of the re-implementation called RDMA-MaaT are shown in Algorithm 6 and 7.

Algorithm 6: RDMA-MaaT (Part 1)

```
1  Function Read(PK, T):
2      do res ← AcqIMExcLock(PK) ;
3      while ¬res;
4      X ← ReadDI(PK);
5      X^m.rL ← {T.Tid } ∪ X^m.rL ;
6      X^m.lock ← 0; WriteDI(PK, X);
7      T^l.rs ← {X}∪ T^l.rs;
8      T^l.aL ← X^m.wL ∪ T^l.aL;
9      T^l.gwts ← max(T^l.gwts, X^m.wts +1);
10 Function Write(PK, T, newV):
11     do res ← AcqIMExcLock(PK) ;
12     while ¬res;
13     X ← ReadDI(PK);
14     X^m.wL ← {T.Tid } ∪ X^m.wL ;
15     X^m.lock ← 0; WriteDI(PK, X);
16     X^d ← newV; T^l.ws ← {X}∪ T^l.ws;
17     T^l.bL ← {X^m.rL } ∪ T^l.bL;
18     T^l.oL ← {X^m.wL } ∪ T^l.oL;
19     T^l.grts ← max(T^l.grts, X^m.rts +1);
20     T^l.gwts ← max(T^l.gwts, X^m.wts +1);
21 Function AcqTMExcLock(Tid):
22     r ← AModIM(Tid, MT_LOCK, 0, EL);
23     return r;
24 Function RlsTMExcLock(Tid):
25     AModIM(Tid, MT_LOCK, EL, 0));
```

Function *Read* is used to do remote read based on *AModIM*, *ReadDI* and *WriteDI*. 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 *ReadDI* 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 ex-

plicitly 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 *AModIM*, *ReadDI* and *WriteDI*. 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 *ReadDI* 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 *AModTM*, *ReadTM* and *WriteTM*. First, we update $T^g.lb$ to $\max\{T^l.grts, T^l.gwts\}$ on the premise of obtaining a remote latch (line 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 3;

$$\begin{cases} T^g.lb = max(T^g.lb, \overline{T}^g.ub + 1) \ //\text{if } \overline{T} \rightarrow T \\ T^g.ub = min(T^g.ub, \overline{T}^g.lb - 1) \ //\text{if } T \rightarrow \overline{T} \end{cases} \quad (3)$$

otherwise, we adjust the timestamp interval of $\overline{T}$ remotely based on Equation 4.

$$\begin{cases} \overline{T}^g.lb = max(\overline{T}^g.lb, T^g.ub + 1) \ //\text{if } T \rightarrow \overline{T} \\ \overline{T}^g.ub = min(\overline{T}^g.ub, T^g.lb - 1) \ //\text{if } \overline{T} \rightarrow T \end{cases} \quad (4)$$

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 18–21); 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 *AModIM*, *ReadDI* and *WriteDI* (line 23–30). 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 31–37). 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*.

Example 6 Consider Figure 8. We present values of metadata including X^d, $X^m.lock$, $X^m.rts$, $X^m.wts$, $X^m.rL$ and $X^m.wL$ at every step of executing RDMA-MaaT in Figure 13. 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 $x^m.lock$ highlighted in red). □

Cicada [31] is a hybrid concurrency control algorithm that imposes OCC on MVCC to achieve serializability. Like MaaT, 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 largest $X_i^m.wts$ smaller than $T^l.bts$, Cicada simply

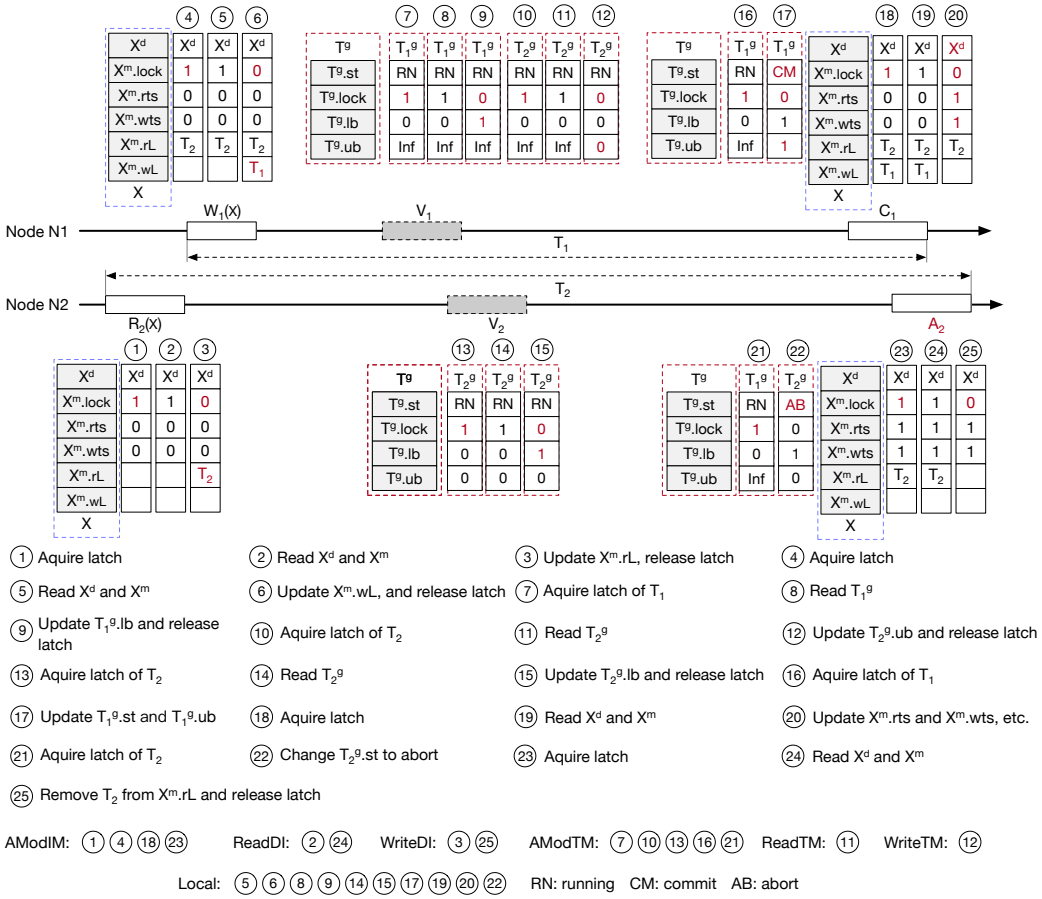


Fig. 13 An example of RDMA-MaaT.

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., $X_i^m.st = AB$), 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. Key functions of the re-implementation called RDMA-Cicada are shown in Algorithm 8.

Function *Read* is used to perform remote read on X based on primitive *ReadDI*. We first perform a remote read of X with all its versions by issuing *ReadDI*. 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 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

Algorithm 7: RDMA-MaaT (Part 2)

```
1  Function Validation(PK, T):
2      do res ← AcqTMExcLock(T.Tid);
3      while ¬res;
4      T^g.lb ← max(T^g.lb, T^l.gwts, T^l.grts );
5      T^g.lock ← 0;
6      foreach Tid∈T^l.bL ∪ T^l.aL ∪ T^l.oL do
7          do res ← AcqTMExcLock(T.Tid);
8          while ¬res;
9          do res ← AcqTMExcLock(Tid);
10         while ¬res;
11         T ←ReadTM(Tid);
12         AdjustTsInterval(T, T);
13         // Equation 3 and 4;
14         Tid^g.lock ← 0;
15         WriteTM(Tid,T); T^g.lock ← 0;
16 Function Commit(T):
17     do res ← AcqTMExcLock(T.Tid);
18     while ¬res;
19     if T^g.lb ≤T^g.ub then
20         T^g.st ← CM; T^g.ub ← T^g.lb;
21     T^g.lock ← 0;
22     if T^g.st ≠ CM then Abort T;
23     foreach X ∈ T^l.ws do
24         do res ← AcqIMExcLock(X.PK) ;
25         while ¬res;
26         X ← ReadDI(PK); X^d ← X^d;
27         X^m.wts ← max(X^m.wts ,T^g.lb );
28         X^m.rts ← max(X^m.rts ,T^g.lb );
29         X^m.wL ← X^m.wL - T.Tid;
30         X^m.lock ← 0; WriteDI(PK, X);
31     foreach X ∈ T^l.rs do
32         do res ← AcqIMExcLock(X.PK) ;
33         while ¬res;
34         X ← ReadDI(PK);
35         X^m.rts ← max(X^m.rts ,T^g.lb );
36         X^m.rL ← X^m.rL - T.Tid;
37         X^m.lock ← 0; WriteDI(PK, X);
```

Algorithm 8: RDMA-Cicada

```
1  Function Read(PK, T):
2      do
3          X ← ReadDI(PK);
4          X_i ← getProperVer(X, T);
5          if X_i = NULL then Abort T;
6      while X_i^m.st = RN;
7      T^l.rs ← {X}∪ T^l.rs;
8  Function Write(PK, T, newV):
9      X←ReadDI(PK); X_k←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 ← {⟨X, newV⟩}∪ T^l.ws;
12 Function Validation(T):
13     foreach ⟨X, newV⟩ ∈ T^l.ws do
14         if ¬AcqIMExcLock(X.PK) then
15             Abort T;
16         X ← ReadDI(X.PK);
17         X_k ← getlatestVer(X);
18         if X_k ≠ X_k then
19             RlsIMExcLock(PK); Abort T;
20         create a new version X_{k+1} of X by setting
             X^d_{k+1} to newV;
21         X^m_{k+1}.st ← RN;
22         X^m_{k+1}.wts ← T^l.bts;
23         X^m.lock ← 0; WriteDI(PK, X);
24     foreach X ∈ T^l.rs - T^l.ws do
25         if ¬AcqIMExcLock(X.PK) then
26             Abort T;
27         X ← ReadDI(X.PK);
28         X_i ← getProperVer(X, T);
29         if X_i ≠ X_i then
30             RlsIMExcLock(PK);Abort T;
31         X^m_i.rts ← max(X^m_i.rts ,T^l.bts );
32         X^m.lock ← 0; WriteDI(PK, X);
33 Function Commit(T):
34     foreach ⟨X, newV⟩ ∈ T^l.ws do
35         AModIM(PK, MT_X_{k+1}_ST, RN, CM);
```

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 $X^m_{k+1}.st$ and $X^m_{k+1}.wts$, and we finally issue *WriteDI* 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 $X^m_i.rts$ and release the latch by issuing *WriteDI* (line 29–32).

Upon commit of T, for each data item X in $T^l.ws$, we set the status $X^m_{k+1}.st$ of X to CM (line 35), where $MT\_X_{k+1}\_ST$ denotes the meta $X^m_{k+1}.st$. Upon abort of T, we set the status $X^m_{k+1}.st$ of X to AB. For illustration purposes, we present how RDMA-Cicada works in Example 7.

Example 7 Consider Figure 8. We present values of metadata including X^d_i, $X^m_i.lock$, $X^m_i.rts$, $X^m_i.wts$ and $X^m_i.st$ at every step of executing RDMA-Cicada in Figure 14. 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 $X^m_1.rts$, $X^m_1.wts$ and $X^m_1.st$ (step ③ – ⑤). In the validation phase of T_2, T_2 acquires the latch on X, re-reads X, sets $X^m_0.rts$, and does a remote update on X (step

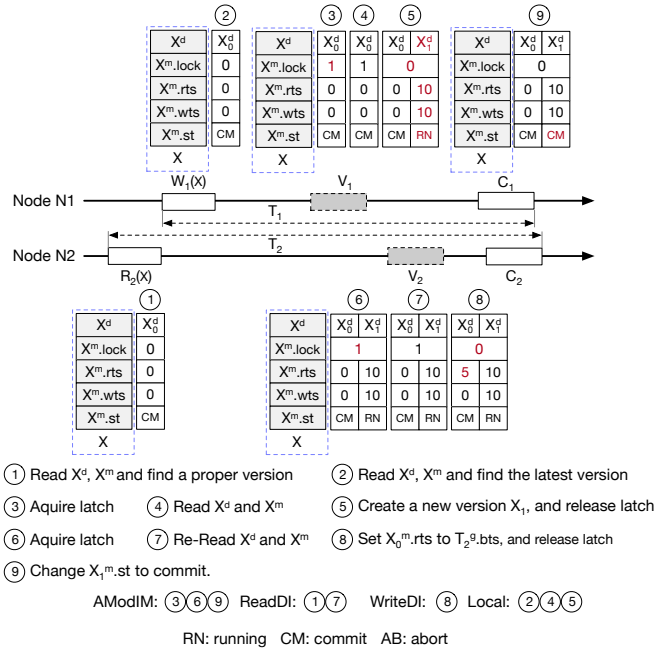


Fig. 14 An example of RDMA-Cicada.

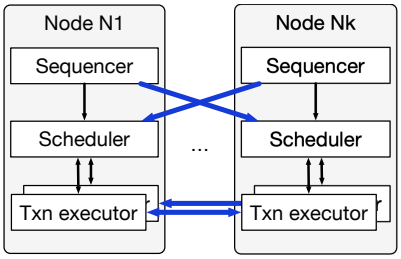


Fig. 15 Calvin overview.

⑥ – ⑧). Finally, T_1 commits, and T_1 sets $X_1^m.st$ to CM using AModIM (step ⑨). □

5.4 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 15. For opportunity one, we implement a circular buffer following FaRM [18] 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 *WriteDI*. After Ts_2 gathers all read sets of remote nodes, Ts_2 can continue to perform execution.
Discussion. However, as reported in [25], 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 [35], LADS [50] and QueCC [36] whose network usage is quite limited.

5.5 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

Table 2 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 %	

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 *ReadDI* with *AModIM* in Function *AcqIMShLock*(line 2–9 in Algorithm 1) because the inputs of AModIM primitive rely on the result of ReadDI 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 3 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 results 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.

6 Experiment

6.1 Setup

We use two popular OLTP benchmarks, YCSB [15] and TPCC [43], to evaluate our re-implementations of concurrency control algorithms.

YCSB [15] 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. 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. **By default, we set both the write ratio and the skew factor to 0.2.**

TPCC [43] 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 [25, 52], we focus on Payment and New-order only to evaluate the transaction scalability of distributed transactions.

We evaluate our system on 4 nodes of an RDMA-capable EDR cluster. Each node is equipped with one Intel(R) Xeon(R) Gold 5220 CPU @ 2.20GHz (18 cores×2

Table 3 A comparison of one-sided implementation with different optimization

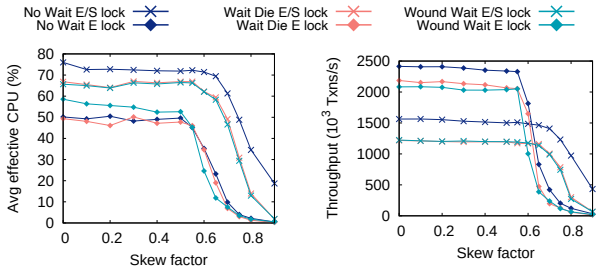
Algorithm	one-sided		+ db&pa&or		+ coroutine		+ cor+db&pa&or	
	TPS(k)	$\overline{\mathcal{U}^e_{CPU}}$	TPS(k)	$\overline{\mathcal{U}^e_{CPU}}$	TPS(k)	$\overline{\mathcal{U}^e_{CPU}}$	TPS(k)	$\overline{\mathcal{U}^e_{CPU}}$
No-Wait	991.1	15 %	1091.0	21 %	**2274.3**	78 %	2145.6	58 %
Wait-Die	807.8	14 %	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 %

HT) processor, 128GB RAM, and one ConnectX-5 EDR 100Gb/s InfiniBand MT27800. By default, each node is configured to have 24 threads, and each thread is set to have 8 coroutines.

6.2 Effect of RDMA networks

We study the effect of RDMA networks on YCSB in terms of throughput (TPS) and $\overline{\mathcal{U}^e_{CPU}}$. 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 [25] by two-sided verb invocations.

The results are reported in Table 2. 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 Calvin. Specifically, Silo performs the best. The reason is that the throughput is closely related to the averaged number (σ) of primitive calls per transaction. For reference, we report σ for each algorithm in Table 2, 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. 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.



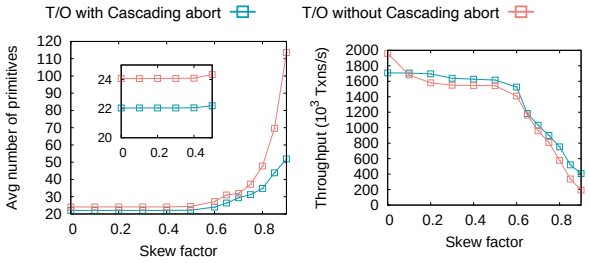
(a) $\overline{\mathcal{U}^e_{CPU}}$ (b) TPS

Fig. 16 Effect of different lock types

6.3 Effect of Various Optimizations

6.3.1 General Optimizations

We investigate the effect of applying optimizations including coroutine, OR, DB, and PA. Table 3 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 coroutine reduces the idle time of CPU, which dominates the benefit brought by DB, PA, and OR. Again, Silo performs the best, followed by No-Wait and MVCC.



(a) Number of primitive calls (b) TPS

Fig. 17 Effect with or without cascading abort

6.3.2 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 16. When the skew factor varies from 0 to 0.6, $\overline{\mathcal{U}^e_{CPU}}$ remains stable for all 2PL variants. Interestingly, variants under *E lock* perform better than those under *E/S lock*. This is because, for two concurrent operations on the same data item, *E lock* simply considers them as conflicts while *E/S lock* requires extra CPU cycles to examine whether they are truly conflicted. Under a low contention scenario, the benefit brought by improving the concurrency is dominated by the overhead of the conflict examination. In contrast, when the skew factor is higher than 0.7, the benefit of improving concurrency is comparable to or even dominates the overhead of the conflict examination. *E/S lock* performs better than *E lock*. *Note, we cannot simply compare the performance of two different algorithms simply based on their* $\overline{\mathcal{U}^e_{CPU}}$.

6.3.3 Variants of T/O

We plot the result of T/O with or without enabling cascading abort in Figure 17. 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 17(a), T/O without cascading abort consumes extra one-sided verb calls to manipulate two pending lists ($X^m.prL$ and $X^m.pwL$), which is comparable to that brought by cascading abort.

6.4 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 18(b) and 18(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 6.3, Cicada performs the best under the high contention, mostly attributable to the multi-version mechanism it uses to enable concurrent read and write operations. Figure 18(d) and 18(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.

6.5 Effect of Write Ratios

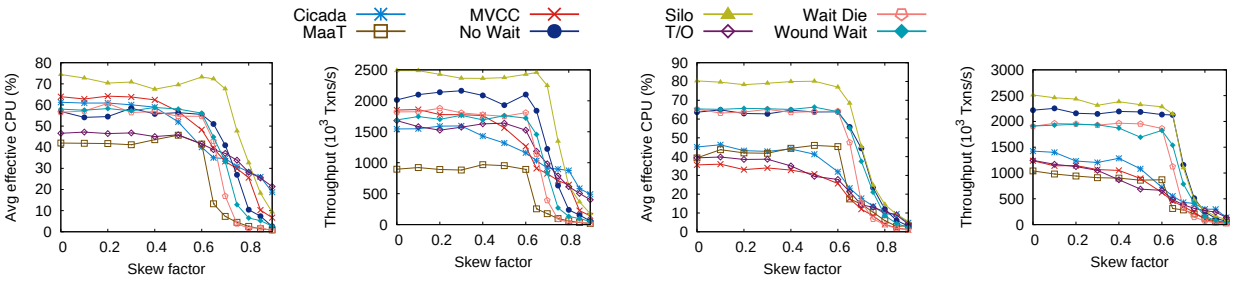
We evaluate the performance by varying write ratios from 0 to 1.0. Figure 19(b) and 19(a) report the throughput and $\overline{\mathcal{U}^e_{CPU}}$ under low contention (skew factor = 0.2), respectively. It can be observed that with increasing the write ratio, the throughput of Silo, Cicada, MVCC, and 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 19(d) and 19(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.

6.6 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 $\mathcal{N}$. In each node of our evaluation, the number of threads is set to 8, and the number of coroutines per thread is set to 4.

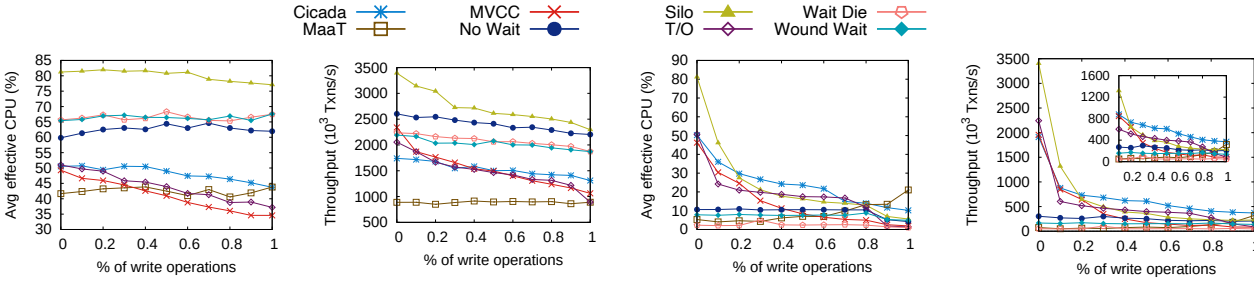
6.6.1 Transaction Scalability

We test the transaction scalability by varying θ from 3 to 15, and fixing $\mathcal{N}$ to 15. Each transaction is composed



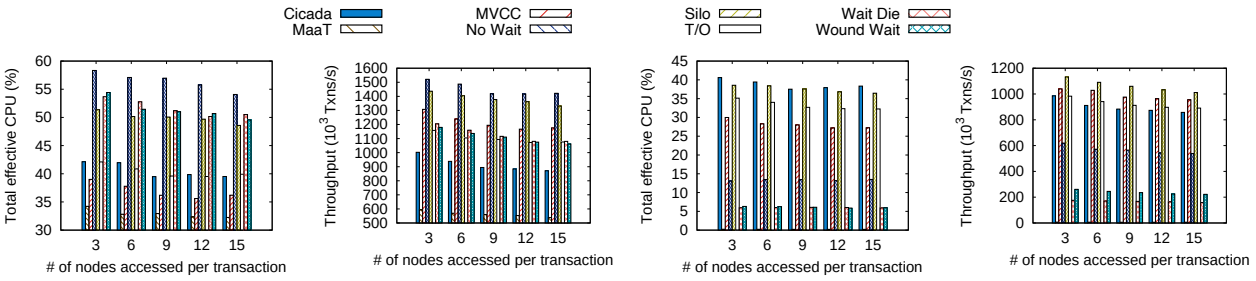
(a) $\overline{\mathcal{U}^e_{CPU}}$(low write-ratio) (b) TPS(low write-ratio) (c) $\overline{\mathcal{U}^e_{CPU}}$(high write-ratio) (d) TPS(high write-ratio)

Fig. 18 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. 19 The comparison of different write-ratios.



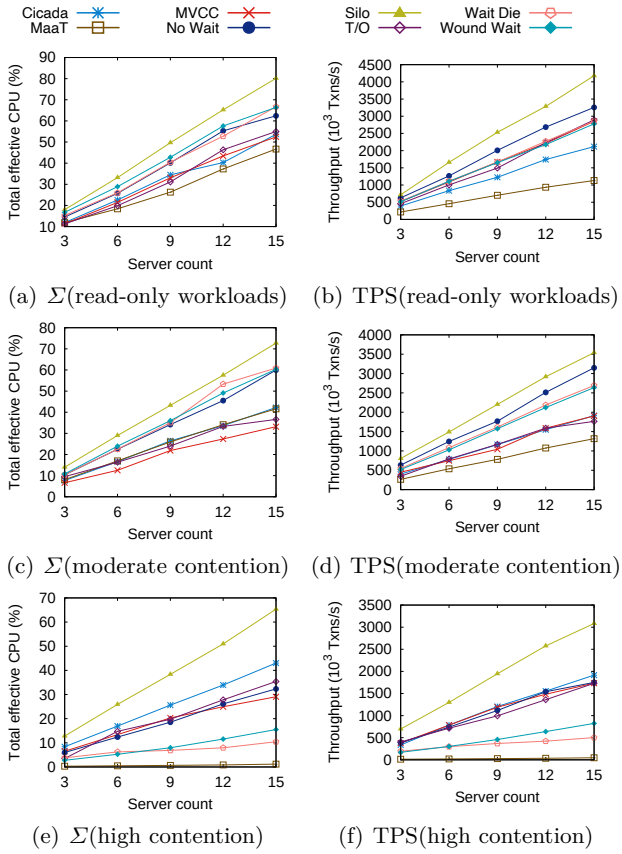
(a) Σ(low contention) (b) TPS(low contention) (c) Σ(high contention) (d) TPS(high contention)

Fig. 20 Transaction scalability

of 15 read/write operations, with the percentage of remote operations fixed to 14/15. We report the throughput and total CPU utilization rate ($\Sigma = \mathcal{N} \times \overline{\mathcal{U}^e_{CPU}}$) under low contention (skew factor = 0.2) in Figure 20(b) and 20(a), respectively. The throughput for different algorithms ranges from 0.6M TPS to 1.5M TPS, and for all algorithms, Σ as well as the throughput drops rather slightly when θ is smaller than 12, while they remain stable when θ is greater than 12. We claim these algorithms can achieve arguably transaction scalability. Note, due to limited resources, our evaluation is deployed on 15 virtual machines. The reason for a slight drop in throughput is that the possibility of operations of a transaction located at the same physical node drops when θ varies, i.e., the number of distributed transactions increases; almost all transactions are distributed when θ is greater than 12, and the latency among different nodes is similar in RDMA network, leading to a stable throughput. We report the results under high contention (skew factor = 0.8) in Figure 20(d) and 20(c), 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.

6.6.2 System Scalability

We evaluate the system scalability by varying $\mathcal{N}$ from 3 to 15 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 21(a), 21(c) and 21(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



(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. 21 System scalability under YCSB.

21(b), 21(d) and 21(f), respectively. The results show that the throughput increase linearly, which follows the same trend with Σ. We plot the results over TPCC in Figure 22(a)–22(d). As we can see, the result follows a similar trend to that over YCSB.

6.7 Comparison with other RDMA-based algorithms

To show the effectiveness of our re-implementations, we compare Silo against DrTM+H [47], another RDMA-based re-implementation of Silo, and compare No Wait, Wait Die and Wound Wait against DSLR[51], another RDMA-based re-implementation of 2PL variants. We report the comparison in Figure 23. 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 calls as Wait-Die and Wound-Wait, while No-Wait consumes the smallest one-sided verbs calls. 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 maintains a fixed size of $X^m.pL$, which causes extra aborts of transactions under the high contention levels.

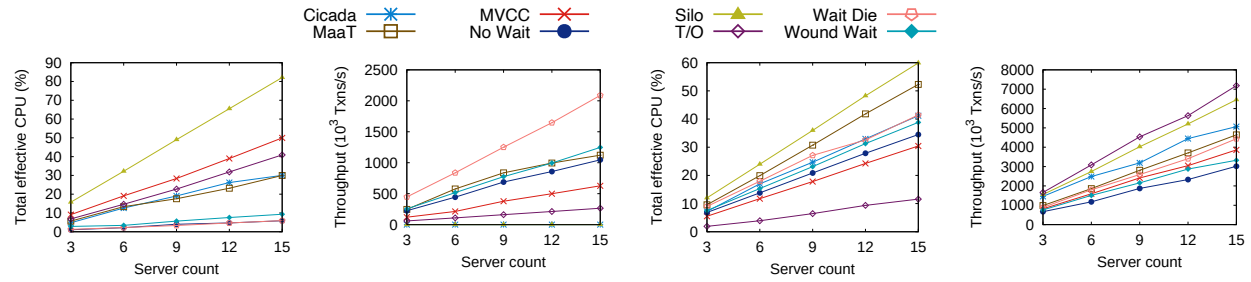
6.8 Summary

By conducting an extensive experimental study, we make the following findings.

1. Transaction scalability can be arguably achieved (Section 6.6.1).
2. Concurrency control algorithms can achieve the system scalability. (Section 6.6.2).
3. For the same algorithm, $\mathcal{U}^e_{CPU}$ is the dominant factor to scale out distributed transaction processing (Section 1 and 6.6).
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 6.7).
5. Optimizing Calvin using RDMA cannot bring obvious benefits (Section 5.4 and 6.2). For other algorithms, RDMA networks bring significant performance improvement, and the degree of improvement closely relies on the number of primitive calls and metadata complexity. Based on these criteria, Silo is reported to be the best in the most cases (Section 6.2, 6.4 and 6.5).
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 6.3.1 and 6.3.2).
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 6.3.3).

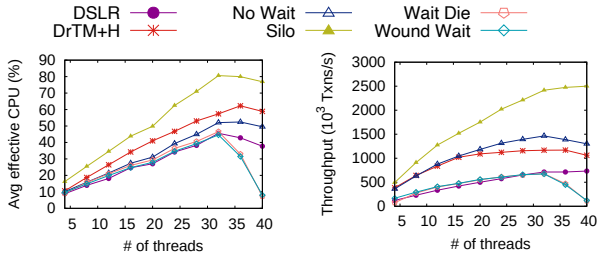
7 Related Work

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) trans-



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

Fig. 22 System scalability under TPCC.



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

Fig. 23 Comparison with RDMA-based algorithms [47,51].

action 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 [1,7,11,41,10] 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 [29,27, 21,52,13,49,40,28]. Recently, there is an increasing interest in evaluating algorithms for distributed transaction processing [11,12,25]. 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 [22,17,20,2,33,16, 30,38,39,55] 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. The majority of them are mainly tailored for some particular concurrency control algorithms. Dragojevi et. al. [19,18] implement a distributed in-memory database called FaRM and leverage one-sided verbs to optimize Silo. DrTM+H [47] also re-implements Silo and additionally proposes optimizations based on hybrid one-sided and two-sided verbs. NAM-DB [8] optimizes the snapshot isolation algorithm. A few works [48, 14,3,51] concentrate on optimizations of 2PL algorithms. More related to our work, Wang et al. [45] 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 the transaction scalability under serializable isolation level. We explore the dominant factors to scale out distributed transaction processing, and propose a general framework RCBench based on which it is practical and convenient to re-implement concurrency control algorithms that enjoy all benefits of RDMA networks.

8 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 over the transaction scalability of our implementations. We list a few findings that have not yet been reported elsewhere. We are confident that these findings pro-

vide a potential guideline to develop highly scalable distributed databases.

References

1. Agrawal, R., Carey, M.J., Livny, M.: Concurrency control performance modeling: Alternatives and implications. ACM Trans. Database Syst. **12**(4), 609–654 (1987)
2. Agrawal, S., Narasayya, V.R., Yang, B.: Integrating vertical and horizontal partitioning into automated physical database design. In: SIGMOD Conference, pp. 359–370. ACM (2004)
3. Barthels, C., Müller, I., Taranov, K., Alonso, G., Hoefler, T.: 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)
4. Berenson, H., Bernstein, P.A., Gray, J., Melton, J., O'Neil, E.J., O'Neil, P.E.: A critique of ANSI SQL isolation levels. In: SIGMOD Conference, pp. 1–10. ACM Press (1995)
5. Bernstein, P.A., Goodman, N.: Timestamp-based algorithms for concurrency control in distributed database systems. In: VLDB, pp. 285–300. IEEE Computer Society (1980)
6. Bernstein, P.A., Goodman, N.: Concurrency control in distributed database systems. ACM Comput. Surv. **13**(2), 185–221 (1981)
7. Bhide, A., Stonebraker, M.: A performance comparison of two architectures for fast transaction processing. In: International Conference on Data Engineering (1988)
8. Binnig, C., Crotty, A., Galakatos, A., Kraska, T., Zamanian, E.: The end of slow networks: It's time for a redesign. Proc. VLDB Endow. **9**(7), 528–539 (2016)
9. Boksenbaum, C., Cart, M., Ferrié, J., Pons, J.: Certification by intervals of timestamps in distributed database systems. In: VLDB (1984)
10. Carey, M.J.: An abstract model of database concurrency control algorithms. In: SIGMOD Conference, pp. 97–107. ACM Press (1983)
11. Carey, M.J., Livny, M.: Distributed concurrency control performance: A study of algorithms, distribution, and replication. In: Fourteenth International Conference on Very Large Data Bases (1988)
12. Carey, M.J., Livny, M.: Parallelism and concurrency control performance in distributed database machines. In: SIGMOD Conference, pp. 122–133. ACM Press (1989)
13. Carey, M.J., Muhanna, W.A.: The performance of multiversion concurrency control algorithms. ACM Trans. Comput. Syst. **4**(4), 338–378 (1986)
14. Chen, Y., Wei, X., Shi, J., Chen, R., Chen, H.: Fast and general distributed transactions using RDMA and HTM. In: EuroSys, pp. 26:1–26:17. ACM (2016)
15. Cooper, B.F., Silberstein, A., Tam, E., Ramakrishnan, R., Sears, R.: Benchmarking cloud serving systems with YCSB. In: SoCC, pp. 143–154. ACM (2010)
16. Curino, C., Zhang, Y., Jones, E.P.C., Madden, S.: Schism: a workload-driven approach to database replication and partitioning. Proc. VLDB Endow. **3**(1), 48–57 (2010)
17. Das, S., Nishimura, S., Agrawal, D., Abbadi, A.E.: Albatross: Lightweight elasticity in shared storage databases for the cloud using live data migration. Proc. VLDB Endow. **4**(8), 494–505 (2011)
18. Dragojevic, A., Narayanan, D., Castro, M., Hodson, O.: Farm: Fast remote memory. In: NSDI, pp. 401–414. USENIX Association (2014)
19. Dragojevic, A., Narayanan, D., Nightingale, E.B., Renzelmann, M., Shamis, A., Badam, A., Castro, M.: No compromises: distributed transactions with consistency, availability, and performance. In: SOSP, pp. 54–70. ACM (2015)
20. Elmore, A.J., Arora, V., Taft, R., Pavlo, A., Agrawal, D., Abbadi, A.E.: Squall: Fine-grained live reconfiguration for partitioned main memory databases. In: SIGMOD Conference, pp. 299–313. ACM (2015)
21. Elmore, A.J., Das, S., Agrawal, D., Abbadi, A.E.: Zephyr: live migration in shared nothing databases for elastic cloud platforms. In: SIGMOD Conference, pp. 301–312. ACM (2011)
22. Elmore, A.J., Das, S., Agrawal, D., El Abbadi, A.: Towards an elastic and autonomic multitenant database. In: Proc. of NetDB Workshop. sn (2011)
23. Gray, J., Lorie, R.A., Putzolu, G.R., Traiger, I.L.: Granularity of locks and degrees of consistency in a shared data base. In: Readings in database systems (3rd ed.) (1976)
24. Härder, T.: Observations on optimistic concurrency control schemes. Inf. Syst. **9**(2), 111–120 (1984)
25. Harding, R., Aken, D.V., Pavlo, A., Stonebraker, M.: An evaluation of distributed concurrency control. PVLDB **10**(5), 553–564 (2017)
26. Hemmatpour, M., Montrucchio, B., Rebaudengo, M., Sadoghi, M.: Analyzing in-memory nosql landscape. IEEE Trans. Knowl. Data Eng. **34**(4), 1628–1643 (2022)
27. Huang, J., Stankovic, J.A., Ramamritham, K., Towsley, D.F.: Experimental evaluation of real-time optimistic concurrency control schemes. In: VLDB, pp. 35–46. Morgan Kaufmann (1991)
28. Huang, Y., Qian, W., Kohler, E., Liskov, B., Shrira, L.: Opportunities for optimism in contended main-memory multicore transactions. Proc. VLDB Endow. **13**(5), 629–642 (2020)
29. Jipping, M.J., Ford, R.: Predicting performance of concurrency control designs. In: SIGMETRICS, pp. 132–142. ACM (1987)
30. Jones, E.P.C.: Fault-tolerant distributed transactions for partitioned OLTP databases. Ph.D. thesis, Massachusetts Institute of Technology, Cambridge, MA, USA (2012)
31. Lim, H., Kaminsky, M., Andersen, D.G.: Cicada: Dependably fast multi-core in-memory transactions. In: SIGMOD Conference, pp. 21–35. ACM (2017)
32. Mitchell, C., Geng, Y., Li, J.: Using one-sided RDMA reads to build a fast, cpu-efficient key-value store. In: USENIX Annual Technical Conference, pp. 103–114. USENIX Association (2013)
33. Pavlo, A., Curino, C., Zdonik, S.B.: Skew-aware automatic database partitioning in shared-nothing, parallel OLTP systems. In: SIGMOD Conference, pp. 61–72. ACM (2012)
34. Peng, D., Dabek, F.: Large-scale incremental processing using distributed transactions and notifications. In: OSDI, pp. 251–264. USENIX Association (2010)
35. Qadah, T., Gupta, S., Sadoghi, M.: Q-store: Distributed, multi-partition transactions via queue-oriented execution and communication. In: EDBT, pp. 73–84. OpenProceedings.org (2020)
36. Qadah, T.M., Sadoghi, M.: Quecc: A queue-oriented, control-free concurrency architecture. In: Middleware, pp. 13–25. ACM (2018)
37. Rosenkrantz, D.J., Stearns, R.E., II, P.M.L.: System level concurrency control for distributed database systems. ACM Trans. Database Syst. **3**(2), 178–198 (1978)
38. Schiller, O., Cipriani, N., Mitschang, B.: Prorea: live database migration for multi-tenant RDBMS with snapshot isolation. In: EDBT, pp. 53–64. ACM (2013)

39. Shute, J., Vingralek, R., Samwel, B., Handy, B., Whipkey, C., Rollins, E., Oancea, M., Littlefield, K., Menestrina, D., Ellner, S., Cieslewicz, J., Rae, I., Stancescu, T., Apte, H.: F1: A distributed SQL database that scales. Proc. VLDB Endow. **6**(11), 1068–1079 (2013)
40. Tanabe, T., Hoshino, T., Kawashima, H., Tatebe, O.: An analysis of concurrency control protocols for in-memory databases with ccbench. Proc. VLDB Endow. **13**, 3531 – 3544 (2020)
41. Thomasian, A.: Concurrency control: Methods, performance, and analysis. ACM Comput. Surv. **30**(1), 70–119 (1998)
42. Thomson, A., Diamond, T., Weng, S., Ren, K., Shao, P., Abadi, D.J.: Calvin: fast distributed transactions for partitioned database systems. In: SIGMOD Conference, pp. 1–12. ACM (2012)
43. TPC-C: http://www.tpc.org/tpcc/ (1988)
44. Tu, S., Zheng, W., Kohler, E., Liskov, B., Madden, S.: Speedy transactions in multicore in-memory databases. In: SOSP, pp. 18–32. ACM (2013)
45. Wang, C., Qian, X.: Rdma-enabled concurrency control protocols for transactions in the cloud era. IEEE Transactions on Cloud Computing **PP**, 1–1 (2021)
46. Wang, T., Kimura, H.: Mostly-optimistic concurrency control for highly contended dynamic workloads on a thousand cores. Proc. VLDB Endow. **10**(2), 49–60 (2016)
47. Wei, X., Dong, Z., Chen, R., Chen, H.: Deconstructing rdma-enabled distributed transactions: Hybrid is better! In: OSDI, pp. 233–251. USENIX Association (2018)
48. Wei, X., Shi, J., Chen, Y., Chen, R., Chen, H.: Fast in-memory transaction processing using RDMA and HTM. In: SOSP, pp. 87–104. ACM (2015)
49. Wu, Y., Arulraj, J., Lin, J., Xian, R., Pavlo, A.: An empirical evaluation of in-memory multi-version concurrency control. Proc. VLDB Endow. **10**(7), 781–792 (2017)
50. Yao, C., Agrawal, D., Chen, G., Lin, Q., Ooi, B.C., Wong, W., Zhang, M.: Exploiting single-threaded model in multi-core in-memory systems. IEEE Trans. Knowl. Data Eng. **28**(10), 2635–2650 (2016)
51. Yoon, D.Y., Chowdhury, M., Mozafari, B.: Distributed lock management with RDMA: decentralization without starvation. In: SIGMOD Conference, pp. 1571–1586. ACM (2018)
52. Yu, X., Bezerra, G., Pavlo, A., Devadas, S., Stonebraker, M.: Staring into the abyss: An evaluation of concurrency control with one thousand cores. Proc. VLDB Endow. **8**(3) (2014)
53. Zamanian, E., Binnig, C., Kraska, T., Harris, T.: The end of a myth: Distributed transactions can scale. CoRR **abs/1607.00655** (2016)
54. Zamanian, E., Yu, X., Stonebraker, M., Kraska, T.: Rethinking database high availability with RDMA networks. Proc. VLDB Endow. **12**(11), 1637–1650 (2019)
55. Zhao, Z.: Efficiently supporting adaptive multi-level serializability models in distributed database systems. In: SIGMOD Conference, pp. 2908–2910. ACM (2021)