RDMA-enabled Concurrency Control Protocols for Transactions in the Cloud Era

Chao Wang, University of Southern California, and Xuehai Qian, University of Southern California

Abstract—Online transaction processing (OLTP) is widely used on modern cloud infrastructures to complete important businesses such as payments and stock exchanges. Remote Direct Memory Access (RDMA) is a technology that enables ultra-low inter-server memory access latency - which is critical for implementing high-performance concurrency control protocols in distributed OLTP. In this paper, we develop RCC, the first unified and comprehensive RDMA-enabled distributed transaction processing framework containing six serializable concurrency control protocols—not only the classical protocols NOWAIT, WAITDIE, OCC, but also more advanced MVCC, SUNDIAL, and CALVIN — the deterministic protocol. Our goal is to unbiasedly compare protocols on OLTP workloads in a common execution environment with the concurrency control protocol being the only changeable component. From RCC, we obtained new insights on building RDMA-based protocols. We analyzed stage-wise latency breakdown to develop more efficient hybrid implementations. Moreover, RCC can enumerate all stage-wise hybrid designs under a given workload characteristic. Our results show that throughput-wise hybrid designs are better than RPC or one-sided counterparts by 32.2% and up to 67%; three hybrid designs are better than their pure counterparts by up to 17.8%. RCC can provide performance insights and be used as the common open-sourced infrastructure for fast prototyping new implementations.

Index Terms—OLTP, Distributed Database Systems. Concurrency Control.

1 Introduction

Online transaction processing (OLTP) has ubiquitous applications in many important domains, including banking, stock marketing, e-commerce, etc. Many cloud infrastructure providers such as Microsoft Azure, Amazon Web Services and Google Cloud Platform have established their cloud database services to support OLTP. With data volume growing exponentially, the cloud database is partitioned and managed by different instance servers. Upon receiving a request, each server is responsible for accessing a subset of the database. Partitioning data such that all queries of one request access only one partition is challenging [1], [2]. Therefore, a transaction inevitably has to access multiple partitions in a distributed manner.

Distributed transactions should guarantee two key properties: (a) atomicity: either all or none of the machines agree to apply the updates; and (2) serializability: all transactions must commit in some serializable order. To ensure these properties, concurrency control protocols have been investigated for decades [3], [4], [5], [6], [7], [8]. The challenge of multi-partition serializable concurrency control protocols is the significant performance penalty due to communication and coordination among distributed machines [9], [10], [11]: when a transaction accesses multiple records over the network, it needs to be serialized with all conflicting transactions [12], making a high-performance network critical.

Remote Direct Memory Access (RDMA) is a technology that enables the network interface controller (NIC) to access the memory of remote servers. It can be used in many domains such as data prefetching [13]. Due to its high bandwidth and low latency, RDMA has been recently used to support distributed transaction systems [14], [15], [16], [17], [18], and has enhanced the performance by orders of magnitude compared to traditional systems using TCP. RDMA network supports both TCP-like *two-sided* communication using primitives `SEND/RECV`, and *one-sided* communication using primitives `READ/WRITE/ATOMIC`, which are capable of accessing remote memory while bypassing traditional network stack, the kernel, and the remote CPUs.

Extensive studies have been conducted in understanding the performance implication of each primitive using micro-benchmarks [14], [18], [19], [20], [21]. Moreover, RDMA has been used to implement the Optimistic Concurrency Control (OCC) protocol [16], [19], [20]. Two takeaways from DrTM+H [20] are: (1) the best performance of OCC cannot be simply achieved by solely using two-sided or one-sided communication; and (2) different communication primitives are best suited for each execution stage.

We claim that the state-of-the-art RDMA-based system DrTM+H [20] is *not* sufficient for two important reasons. First, in real-world applications, various concurrency control protocols [7], [22], [23], [24], [25], [26], [27] are used, the understanding of RDMA implications on OCC may *not transfer* to other protocols. Second, building the standalone framework for each individual protocol does not allow the fair and unbiased *cross-protocol* performance comparison. In a complete system for distributed transaction execution, concurrency control protocol is only one component, the system organization, optimizations, and transaction execution model can vary a lot. Having a common execution environment for all various protocols is critical to draw any meaningful conclusions [28], [29]. Clearly, DrTM+H does not provide such capability. Compared to DrTM+H, Deneva [30] studied six concurrency control protocols based on TCP, affirming the importance of cross-protocol comparison. However, Deneva is not based on RDMA.

In this paper, we take the important step to close the gap. We develop *RCC*, the *first* unified and comprehensive RDMA-enabled distributed transaction processing frame-

work supporting multiple concurrency control protocols with different properties. It includes protocols in a wide spectrum: (1) classical protocols such as two-phase-locking (2PL), i.e., NOWAIT [4], WAITDIE [4], and OCC [6]; (2) more advanced protocols such as MVCC [5], which has been adopted by many modern cloud database system providers, and the recent SUNDIAL [31], that allows dynamically adjustment of commit order with logical lease to reduce abort; and (3) CALVIN [11], a shared-nothing protocol that ensures deterministic transaction execution.

RCC enables us to perform *unbiased and fair* comparison of the protocols in a *common* execution environment with the concurrency control protocol being the only changeable component. We develop the *correct and efficient* RDMA-based implementation using known techniques, i.e., co-routines, outstanding requests, and doorbell batching, with two-sided and one-sided communication primitives. To validate the benefits of RDMA, RCC also provides reference implementations based on TCP. As a common infrastructure for RDMA-enabled transaction execution, RCC allows the fast prototyping of existing protocols or *new implementations*.

With RCC, we are able to answer several key questions that cannot be answered before. First, while the protocol specifications are known, we answer the question of *how* to leverage RDMA to construct different protocols with concrete, executable, and efficient implementations. Second, we can perform both apple-to-apple *cross-protocol* comparisons and the *stage-level same-protocol* study on performance and various execution characteristics in the context of the same system organization. The answers of which primitives being best suited for which execution stage can be used to further optimize the performance. Third, for CALVIN , which is a shared-nothing protocol and has never been studied in the context of RDMA, we answer the question of whether the one-sided primitives would bring the similar benefits as other shared-everything protocols.

The implementation of the current RCC with the six protocols has around 25,000 lines of codes written in C++. We intend to open-source the framework in the near future. We try our best to fairly optimize the performance of each without bias using known techniques such as co-routines [16], outstanding requests [20], doorbell batching [32]. We evaluate all protocol designs on a cluster with ConnectX-4 EDR InfiniBand RDMA support using three typical workloads: SmallBank [33], TPC-C [34], and YCSB [35]

Using RCC, we conduct the *first cross-protocol* performance comparison with RDMA and observe a number of interesting performance characteristics: 1) OCC does not always achieve the best performance. In fact, the simple 2PL protocols such as NOWAIT and WAITDIE outperform other more complicated protocols with a high performance RDMA network on a network-intensive workload (SmallBank). This indicates the promise of enhancing RDMA network stack instead of fine-tuning CC protocols in future transaction processing systems.

We obtain the execution stage latency breakdowns with one-sided and two-sided primitive for each protocol for all three workloads, and they are analyzed to develop *hybrid* implementations, which achieves equivalent or better performance under the given workload characteristics. Our experiment shows that by cherry-picking the communication type that incurs lower latency for each protocol stage, we can find new protocol implementations that reaches at most 17.8% throughput speedup, compared to the better side of RPC or one-sided implementations.

To provide user-friendly interface, we designed and implemented simple interface in RCC that allows both common and advanced users to quickly evaluate any hybrid implementation for an existing or new protocol given a workload characteristic. In addition, for a given protocol, RCC can exhaustively enumerate all combinations of hybrid protocols and provide substantial evidence that a certain hybrid design is the best among all possibilities when varying stage communication styles. Our experiments show that throughput-wise hybrid designs are better than RPC or one-sided counterparts by 32.2% on average and up to 67%.

We believe that RCC is a significant advance over state-of-the-art as it can both provide performance insights and be used as the common infrastructure for fast prototyping new implementations.

2 RDMA Basics

Remote Direct Memory Access (RDMA) is a network technology featuring high bandwidth and low latency data transfer with low CPU overhead. It is particularly suitable for large data centers. RDMA operations, i.e., **verbs**, can be classified into two types: (1) *two-sided* primitives `SEND/RECV`; and (2) *one-sided* primitives `READ/WRITE/ATOMIC`. The latter provides the unique capability to directly access the memory of remote machines without involving remote CPUs. This feature makes one-sided operations suitable for distributed applications with high CPU utilization. Although having similar semantics with TCP's send/receive over bound sockets, RDMA two-sided operations bypass the traditional network stack and the OS kernel, making the performance of RPC implementation over RDMA much higher than that over TCP.

To perform RDMA communication, queue pairs (QPs) must be set up. A QP consists of a send queue and a receive queue. When a sender posts a one-sided RDMA request to the send queue, the local QP will transfer data to some remote QP, and the sender can poll for completion information from the completion queue associated with the QP. The receiver's CPU is not aware of the one-sided operations performed by the receiver's RNIC without checking the changes in memory. For a sender to post a two-sided operation, the receiver QP has to post `RECV` for the corresponding `SEND` in advance. It polls the receive queue to obtain the data. To set up a reliable connection, a node has to maintain at least a cluster-size number of QPs in its RDMA-enabled NIC (RNIC), each connected with one remote node.

3 RCC System Organization

3.1 Overall Architecture

Figure 1 shows the overview of RCC, which runs on multiple symmetric distributed nodes, each containing a configurable number of server threads to process transactions. A client thread sends transaction requests to a random local or remote transaction processing thread in the cluster. The stats thread is used to collect the statistics (e.g., the number of committed transactions) generated by each processing thread. The QP thread is used to bootstrap RDMA connections by establishing the QP pairing by TCP connections.

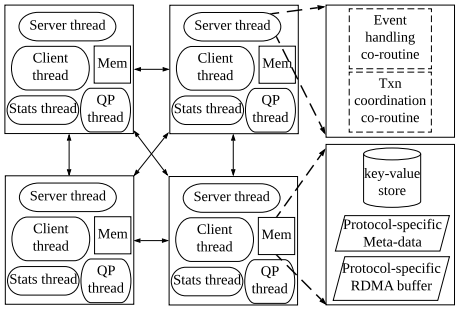


Fig. 1. RCC Framework Overview

RCC uses co-routines as an essential optimization technique [16] to hide network latency. Specifically, each thread starts an *event handling* co-routine and some *transaction coordination* co-routines. An event handling co-routine continuously checks and handles network-triggered events such as polled completions or memory-triggered events such as the release of a lock. A transaction coordination co-routine is where a transaction logically executes.

In RCC, the distributed in-memory database is implemented as a distributed key-value store that can be accessed either locally or remotely via a key and table ID. we leveraged DrTM+H's [20] key-value store as RCC's back-end. Since RCC supports multiple protocols, each protocol has its protocol-specific metadata and RDMA buffer to ensure the correct execution leveraging RDMA primitives.

3.2 Transaction Execution Model

RCC employs a symmetric model to execute transactions: each node serves as both a client and a transaction processing server. As shown in Figure 1, each transaction coordination co-routine is responsible for executing a transaction at any time. We use *coordinator* to refer to the co-routine that receives transaction requests from some local or remote client thread and orchestrates transactional activities in the cluster. We use *participant* to refer to a machine where there is a record to be accessed by some transaction. When a participant receives an RPC request, its event handling co-routine will be invoked to process the request locally. When a participant receives an RDMA one-sided operation, its RNIC is responsible for accessing the memory without interrupting the CPU.

In RCC, A *record* refers to the actual data; and a *tuple* refers to a record associated with the relevant metadata. All tuples are located in RDMA-registered memory. A distributed in-memory key-value store keeps all tuples partitioned among all machines. Since one-sided operations can only access remote memory by leveraging the precomputed remote offsets, to reduce the number of one-sided operations involved in retrieving metadata, the metadata are placed physically together with the record as shown in Figure 3. Currently, RCC only supports fixed record size and variable-sized record can be supported by placing an extra pointer in the record field pointing to an RDMA-registered region, similar to [36]. With one-sided `READ`, the remote offset of a tuple is fetched before the actual tuple is fetched and the offset is then cached locally to avoid unnecessary one-sided operations.

A transaction has a *read set (RS)* and a *write set (WS)* that are known before the execution. The records in RS are read-only. The execution of a transaction is conceptually divided into three primary stages: *fetching*: get the tuples of records in RS and WS, the metadata is used for protocol operations; *execution*: a transaction performs the actual computation locally using the fetched record; and *commit*: a transaction checks if it is serializable, if so, *logs* all writes to remote backup machines for high availability and recovery, and *updates* remote records. Our implementations apply to transactions with one or more fetching and execution stages.

3.3 RDMA Communication and Optimizations

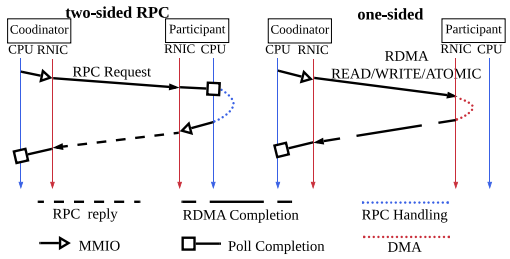


Fig. 2. Two-sided versus one-sided

We use one-sided RDMA primitives like `READ`, `WRITE`, `ATOMIC` with RC QPs and two-sided RDMA primitives like `SEND` and `RECEIVE` over UD QPs to implement RCC. From [16], two-sided primitives over UD QPs outperform one-sided primitives in symmetric transaction systems, and UD mode is much more reliable than expected with RDMA network's lossless link layer. [20] further confirms the unsuitability of one-sided primitives to implement fast RPC.

Figure 2 illustrates the two types of communications in RCC employed by each concurrency control protocol. In two-sided RPC, a coordinator first sends a memory-mapped IO (MMIO) to the RNIC, which in turn `SEND`s an RPC request to the receiver's RNIC. After the corresponding participant `RECV`s the request, its CPU polls a completion event, which later triggers a pre-registered handler function to process the request and send back a reply using similar verbs. In one-sided communication, after the participant receives a one-sided request, i.e., `READ`, `WRITE`, `ATOMIC`, its RNIC will access local memory using a Direct Memory Access (DMA). The completion is signaled when the coordinator polls if it is interested in the completion event.

MMIO is an expensive operation to notify RNIC of a request fetching event. Using *one* MMIO for a batch of RDMA requests can effectively save PCIe bandwidth and improve the performance of transaction systems [20], which is called *doorbell batching*. Meanwhile, having multiple outstanding requests on the fly can save the waiting time of request completion, thus reducing the latency of remote transactions [20]. Leveraging co-routines serve to interleave RDMA communication with computation. RCC uses similar techniques as important optimizations.

4 RDMA-Based Concurrency Control

In RCC, we implement six concurrency control (CC) protocols with two-sided and one-sided RDMA primitives. Among these protocols, NOWAIT [4] and WAITDIE [4] are examples of two-phase locking (2PL) [4] CC algorithms. They differ in conflict resolution,i.e., how conflicts are resolved to ensure serialization. Compared to 2PL, Optimistic Concurrency Control (OCC) [6] reads records speculatively without locking and validates data upon transaction commits—the only time to use locks. MVCC [5] optimizes the performance

of read-heavy transactions by allowing the read of the correct recently committed records instead of aborting. SUNDIAL [31] leverages the dynamically adjustable logical leases to order transaction commits and reduce aborts. CALVIN [11] introduces determinism with a shared-nothing protocol, demonstrating very different communication behavior.

While the protocols themselves are known, we rethink their correct and efficient implementations in the context of RDMA. Each protocol requires techniques to implement specific protocol requirements, particularly atomic tuple read (for MVCC) and update (for SUNDIAL). We describe two implementations of each protocol: *RPC* version, which mostly uses RPCs enabled by RDMA's two-sided communication primitives; and *one-sided* version, which mainly uses RDMA's unique one-sided communication primitives.

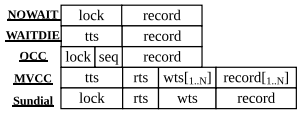


Fig. 3. Metadata

4.1 Transaction Operations

We consider the following common operations used in one or multiple concurrency control protocols. They can be implemented with either RPC or one-sided primitives.

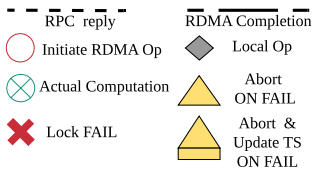


Fig. 4. Legend for RCC

Fetching Tuples are fetched during transaction execution. The read-only records are fetched into RS, other accessed records are fetched into WS.

Locking All RCC protocols need locking to enforce certain logical serialization order. For remote locking, the better implementation choice is affected by the load of remote threads which execute transaction co-routines. The higher load may affect the capability of handling RPC, thus one-sided primitives can be better.

Validation This operation is needed in OCC, MVCC, and SUNDIAL in different stages. The RPC implementation typically requires only one network operation, while the one-sided version may lead to one or more requests. Similar to locking, the best primitive choice is determined by the workload of remote co-routines.

Logging To support high availability and recovery, each protocol logs its updates to some backup servers. Similar to DrTM+H and FaSST, RCC employs coordinator log [37] for two-phase-commit. Only after the successful logging and reception of acknowledgments from all replica, can the transaction writes the updates back to the remote machine. Logs are lazily reclaimed in the background of backup machines when they are notified by the coordinator using two-sided RPC. Logging strongly prefers one-sided `WRITE` to log to backup servers for OCC according to [20]. Our stage-wise latency results support this claim for other protocols.

Update It writes back the updated data and metadata. Two-sided RPCs can finish this update in one round trip; one-sided primitives need two without doorbell batching. The index of the write set entries can be cached in advance to reduce the overhead of this operation when using one-sided primitives. Next, we describe the implementation of each protocol in RCC except for OCC, which is implemented based on DrTM+H [20]. We choose to base our OCC implementation on DrTM+H because it outperforms other RDMA-based OCC implementations by [19] and [16]. Figure 4 shows the legend for protocol operations in this section.

4.2 NOWAIT

NOWAIT [4] is a basic concurrency control algorithm based on 2PL that prevents deadlocks. A transaction in NOWAIT tries to lock all the records accessed; if it fails to lock a record that is already locked by another transaction, the transaction aborts immediately and releases all locks that have been successfully acquired. Figure 5 shows the operations of NOWAIT for both RPC and one-sided implementations.

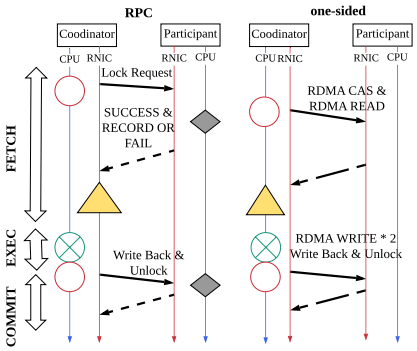


Fig. 5. NOWAIT Implementations

With RPC, a coordinator locks records by sending RPC locking request to the corresponding participant, the RPC handler locks the record using a local CAS. If the CAS fails, a failure message is sent back to the coordinator which will release all read and write locks by posting RPC release requests before aborting the transaction. Otherwise, the participant's handler has already locked the tuple locally, and it returns a success message with the record in response. On transaction commit, with all locks acquired, a write-back request associated with the updated records is sent to each participants, where an RPC handler performs write-back of the record and releases the lock.

With one-sided primitive, we use the doorbell batching mechanism as an efficient way to issue multiple outstanding requests from the sender. With this optimization, only one yield is needed after the last request is posted, thus reducing latency and context switching overhead. On locking, the coordinator needs to perform two operations—RDMA `CAS` and `READ`—to lock and read the remote record. Logically, they should be performed one after another, but in fact, the coordinator can issue `READ` immediately after `CAS` to overlap the communication. It is because the two will be performed in the issue order remotely, and if the lock acquire fails, the coordinator can simply ignore the returned data of `READ`. Note that the read offsets are collected and cached by the coordinator before transaction execution starts and do not incur much overhead. With high contention, the optimization tends to add wasted network traffic. However, for network-intensive applications with low contention, i.e., SmallBank, the throughput increases by 25.1% while average latency decreases by 22.7%. Similarly, two RDMA `WRITE`s are posted to update and unlock the record at the commit stage. Only the second RDMA write is signaled to avoid sending multiple MMIOs and wasting PCIe bandwidth. Different from lock & read, the doorbell batched update & unlock is always beneficial.

Key RCC insight: Doorbell-batched CAS and READ fetching a tuple immediately after the CAS operation, can effectively save network round trips and improve performance.

4.3 WAITDIE

Different from NOWAIT, which unconditionally aborts any transaction accessing conflicting records, WAITDIE resolves

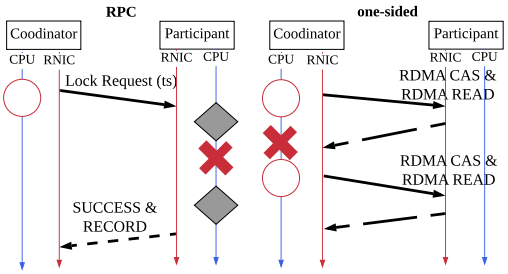


Fig. 6. WAITDIE: the **FETCH** stage

conflicts with a global consensus priority. On start, each transaction obtains a globally unique timestamp, which can be stored in the lock records it accessed. Upon detecting a conflict, the timestamp logged in the lock is compared with the current transaction's timestamp to determine whether to immediately abort the transaction or let it wait. In RCC, we construct the unique timestamp of a transaction by appending the machine ID, thread ID, and coroutine ID to the low-order bits of the local clock time [5]. This avoids the overhead of global clock synchronization as in NTP [38] and PTP [39]. The timestamp is stored in the 64-bit lock record.

Compared to NOWAIT, the new operation in WAITDIE is transaction wait. Figure 6 shows the WAITDIE operations in the fetch stage. With RPC, it can be implemented intuitively: when an accessed record is locked, the lock request handler decides based on the request transaction's timestamp whether to let it wait until unlocked, or send back a failure immediately. Note that the handler does not busy-wait for the lock, which blocks other incoming requests. Instead, the transaction is added to the lock's waiting list, which is checked in the event loop periodically by the handler thread. On lock release, the handler thread removes the transaction from the waiting list and replies to the coordinator with a success message and the locked record.

With one-sided primitive, the implementation is less straightforward. The key difference is that the current transaction needs to obtain the record's timestamp—even if it is locked—and decides to abort or wait by itself. Similar to NOWAIT, we use an RDMA `CAS` followed by an RDMA `READ` to retrieve the remote lock together with its timestamp and record, as seen in Figure 6. If the record is not locked, the CAS succeeds and atomically writes the transaction's timestamp on the remote lock, and returns 0. If the CAS fails,i.e.,the record is locked, rather than abort immediately, the current transaction compares its timestamp with the returned timestamp, which indicates the lock-holding transaction, to determine whether to abort itself or wait. If the decision is to wait, the co-routine keeps posting RDMA `CAS` with `READ` requests and yields after every unsuccessful trial until it succeeds.

Our current one-sided implementation of WAITDIE is not starvation-free for old transactions: when the oldest transaction fails to lock, the lock may be released and reacquired by another younger transaction, making the oldest transaction starve. One potential solution may be that a counter is put along with the timestamp and initialized to be 0. When an old transaction detects failure after the first `CAS` & `Read`, it increments the counter once by issuing an RDMA `FETCH_AND_ADD` operation, all future younger transactions accessing the record will then abort until the counter is reset to 0. Another `FETCH_AND_ADD` is needed to decrement the counter when the old transaction successfully grabs the lock.

Key RCC insight: With the use of co-routines, keep posting doorbell-batched CAS and READ until success does not prevent CPU from processing requests in other co-routines.

4.4 MVCC

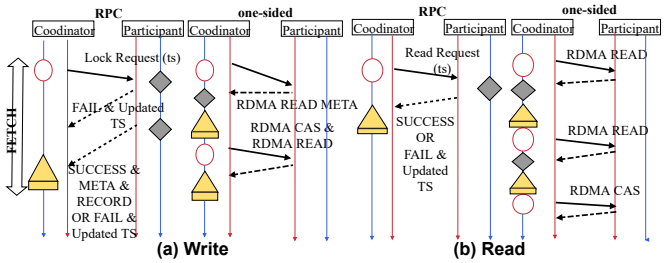


Fig. 7. MVCC

MVCC (Multi-Version Concurrent Control) [5] reduces read-write conflicts by keeping multiple versions of the record and providing a recently committed version when possible. Shown in Figure 3, the metadata of each tuple in MVCC consists of three parts: 1. write lock, which contains the timestamp of the current transaction holding the lock that has not committed yet (`tuple.tts`); 2. read timestamp (`tuple.rts`), which is the latest (largest) transaction timestamp that has successfully read the record; and 3. write timestamps (`tuple.wts`), which are the timestamps of recently *committed* transactions that have performed writes on the record. These versions are kept in the participants. We also denote the timestamp of the current transaction trying to fetch records as `ctts`.

To access a record in RS, we check *Cond R1:* there is a proper record version based on the `tuple.wts` of recently committed transactions—it should choose the largest `tuple.wts` smaller than `ctts`; and *Cond R2:* `tuple.tts` is 0 or larger than `ctts`. Cond R2 means there is no un-committed transaction writing the record, or the write happens *after* the read, in which the read transaction can still correctly gets one of the committed versions of the record. If both Cond R1 and R2 are satisfied, the version from Cond R2 can be returned.

To access a record in WS, we check *Cond W1:* transaction's timestamp is larger than the maximum `tuple.wts` and the current `tuple.rts`; and *Cond W2:* the record is not locked. If either is failed, the transaction is aborted; otherwise, the record is locked with `tuple.tts` updated to `ctts`, a new record is created and sent back.

Conceptually, MVCC maintains the following properties. A write of transaction `ctts` cannot be "inserted" among the committed transactions indicated by `tuple.wts`; and the write should be ordered after any performed read. A read should always return the most recent committed version of a record. The key requirement for correctness is that the condition check for RS and WS record should be *atomic*.

The original MVCC requires using a linked list to maintain a set of record versions. However, the nature of one-sided primitive makes it costly to traverse a remote linked list—in the worst case, the number of one-sided operations for a single remote read is proportional to the number of versions in the list. Thus, we use a static number of memory slots allocated for each record to store the stale versions. A transaction will simply abort when it cannot find a suitable version among the slots available for a read operation. The

number of slots determines the trade-off between the extra read aborts and reduced memory/traversal overhead. We choose four slots because our preliminary experiments show that at most 4.2% of read aborts are due to slot overflow. A typical one-sided MVCC run over SmallBank reads the four version slots in a likelihood of 80% , 13%, 3% and 2%. While our design incurs extra space to allocate a static number of version slots compared to using a linked list, it avoids the more expensive performance penalty for multiple round-trips: 20% of the requests incur more than one round-trips.

In MVCC, we use the same timestamp organization as WAITDIE. The local clock reduces bandwidth overhead of a global clock but may introduce significant bias. While not affecting correctness, the large time gap between different machines may lead to a long waiting time. To mitigate the issue, each transaction co-routine maintains a local time and *adjusts* the local time whenever it finds a larger `tuple.wts` or `tuple.rts` in any tuple received. The encapsulated remote time on the `tuple.wts` or `tuple.rts` is extracted and local time is adjusted accordingly if the extracted remote time is larger. This mechanism limits the gap of local timer between machines, and reduces the chance of abort due to the lack of suitable version among the fixed version slots.

While it is not hard to conceptually understand MVCC, the implementation with RDMA needs to ensure atomicity. Let us first consider accessing records in WS. One way is to first check Cond W2 and lock the record, at this point, the metadata cannot be accessed by other writes, we can reliably check Cond W1. If it is not satisfied, the lock is released and the write transaction aborts. However, in this way we need to perform a lock for every write, even if the write transaction cannot be properly serialized. It is particularly a problem for one-sided primitives, because the lock is implemented with an RDMA `ATOMIC CAS`. The better approach is to first check Cond W2 and then acquire the lock. However, a subtle issue raises because Cond W1 and Cond W2 are not done atomically. Between the point that Cond W1 is satisfied and the point the lock is acquired, another transaction that writes the record can lock the record and commit (unlock). According to the protocol property, the current transaction should be aborted, but it will find both Cond W1 and Cond W2 satisfied. To ensure atomicity while avoiding the overhead of locking. We propose the *double-read* mechanism. After the lock is acquired, Cond W1 should be checked *again*, if it is still satisfied, the write can proceed, otherwise, it is aborted.

As in Figure 7 (a), with RPC, the write protocol can be implemented by the handler on the participant. With one-sided primitive, the coordinator posts an RDMA `READ` to read the metadata of the record—`tuple.rts` and `tuple.wts`—on the participant, then checks Cond W1 locally. If it is satisfied, the coordinator posts an RDMA `ATOMIC CAS` to lock the record, and a second RDMA `READ` to fetch the tuple. Cond W1 can be checked again based on the just returned `tuple.rts` and `tuple.wts`, if it still holds, the returned record `tuple.record` is kept locally in the coordinator. Otherwise, the transaction aborts and the lock is released.

When accessing records in RS, the tuples need to be fetched atomically. The separate double-read mechanism discussed before can be generalized to *two consecutive reads of the same tuple*. If the contents of each returned data are the same, then we are sure that atomicity is not violated. Based on the atomically read tuple, Cond R1 and Cond R2 can be checked to generate the appropriate committed version for the record in RS. If the second tuple returned is different from the first, then the transaction is simply aborted. We apply a small optimization to reduce unnecessary abort: among the two versions of metadata, we only need to ensure the match of `tuple.wts`. The `tuple.tts` can be different since a transaction corresponds to the first `tuple.tts` can be aborted between the two reads. But as long as Cond R2 is satisfied, the read can still get a version among `tuple.wts`.

As in Figure 7 (b), with RPC, the read procedure can be implemented in a straightforward manner with the handler on participant. With one-sided primitives, the two reads are implemented by two doorbell batched RDMA `READ`s. The only additional operation is to use an RDMA `ATOMIC CAS` to update `rts` of the record in the participant. If it fails, we can simply retry until succeed. Note that it does not imply conflict, but just multiple concurrent reads.

On commit, with one-sided primitive, the coordinator locally overwrites the oldest `wts` with its own `ctts`, and updates the corresponding record to the locally created one for write. Then it posts two RDMA `WRITE`s. The first write puts the locally prepared new record+metadata to the participant; the second write releases the lock. With RPC, the procedure can be implemented similarly.

Garbage collection & memory management Since our MVCC uses a static number of slots instead of employing a linked list, all slots are pre-allocated both for the use of two-sided RPC function calls and for one-sided RDMA access. Therefore, it is unnecessary to garbage collect stale versions when they are out of visibility of any read/write.

Clock synchronization MVCC uses local clock plus adjustment instead of global clock synchronization to avoid wasting network bandwidth. Global synchronization protocols like NTP [38] are typically used to keep machines synchronized with the Internet within milliseconds skew. PTP [39] can synchronize network computers within sub-milliseconds skew by employing a Best Master Clock (BMC) algorithm. We integrate the adjustment within the MVCC protocol, making the synchronization on demand.

Key RCC insight: One-sided primitive makes traversing a remote linked list costly; using a static number of version slots can effectively reduce communication overhead.

4.5 SUNDIAL

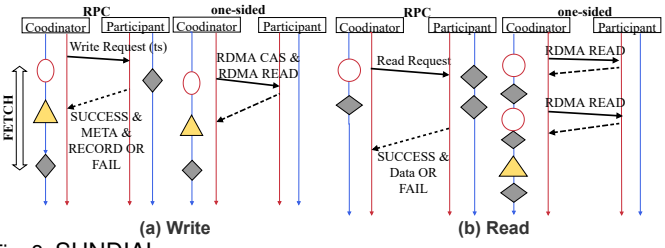


Fig. 8. SUNDIAL

SUNDIAL [31] is an elegant protocol based on logical leases to avoid unnecessary aborts while still maintaining serialization by dynamically adjusting the timestamp of transactions or commit order. Based on the tuple format in Figure 3, the lease of a tuple is specified by `tuple.[wts,rts]`. Each transaction has a `commit_tts`,

which indicates the *required* timestamp of the transaction to satisfy the current lease of accessed records. When accessing a record in RS, the transaction *atomically* reads the tuple and update `commit_tts` to `Max(commit_tts,tuple.wts)`. It is because to correctly read the record, the transaction has to be logically ordered after the most recent writer transaction. When accessing a record in WS, the transaction tries to lock the tuple, and if it is also in RS, checks whether `tuple.wts` is the same as the `RS[key].wts`. The second condition ensures that there is no transaction writing the record committed since the read. If both conditions pass, the transaction's `commit_tts` is updated to `Max(commit_tts,tuple.rts+1)`. It ensures that the transaction is logically ordered after the current lease of the record. Since other transactions may have read the record during the lease, without such update, the transaction would have to be aborted.

Although the update of `commit_tts` during execution will try to satisfy the *current* lease based on *individual* record, at the commit time, the transaction needs to be validated to ensure its current `commit_tts` falls into *all* leases of records in RS. If it is not satisfied, SUNDIAL allows the transaction attempt to *renew* the lease by adjusting the `tuple.rts` in the data store at participant [1]. The renew is failed if (1) the current `wts` is not the same as current `tuple.wts`, meaning that there is a later committed transaction writing the record, which invalidate the previous read record; or (2) the record is locked, meaning that there is a transaction trying to write the record, which prevents the lease extension. Otherwise, the transaction can adjust the lease by updating `tuple.rts` to `commit_tts`. The key requirement is that the lease renewal operation should be performed *atomically*. If all RS records are validated, and all necessary lease renewals are successful, the transaction is committed, updating `tuple.wts` and `tuple.rts` of all records in WS to be `commit_tts`.

For records in WS with one-sided primitives, the tuples can be easily checked after a doorbell batched `CAS` and `READ` to lock and retrieve the tuple, as in Figure 8 (a). Yet to implement SUNDIAL in RCC, we need to solve two problems. First, for records in RS, the tuple needs to be accessed atomically. This can be done using the double doorbell batched reads with one-sided primitives or simply double read with RPC introduced in MVCC, as shown in Figure 8 (b). Second, we need to ensure the atomic lease renewal, which is more challenging than atomic read. To implement this, we first atomically read the tuple from participant, then use an atomic operation to update `tuple.rts`. With these two ideas, we can implement RPC and one-sided SUNDIAL.

In RPC version, the atomic tuple read and lease renewal are all performed by the handler in the participant. The coordinator just poses the read and renewal requests and processes the responses according to the protocol. In one-sided version, the fetch of tuples in RS and WS is similar to MVCC with double doorbell batched reads. Based on the fetched tuples, the coordinator locally performs the SUNDIAL protocol operations. For lease renewal, the coordinator first atomically reads the tuple, then checks the lease extension condition, if it is allowed, it poses an RDMA `ATOMIC CAS` with the previous `tuple.rts` is the old value and its `commit_tts` as the new value. In this way, the lease renewal is performed atomically. It is worth noting that we can implement in this manner because the SUNDIAL protocol only requires updating one variable `tuple.rts` to renew the lease. If multiple variables need to be updated, then more sophisticated mechanisms are needed and it is beyond the scope of the paper.

Key RCC insight: We can efficiently implement atomic tuple read by using double doorbell batched READs of one-sided primitives or double read with RPC. Atomically renew the lease in SUNDIAL by atomically updating `tuple.rts`.

4.6 CALVIN

Different than all other protocols, CALVIN [11] enforces a deterministic order among transaction in an epoch, e.g., all transactions received by the system during a certain time period. The readers can reference the original paper for the complete motivation and advantages of this approach, we are interested in how the communication happen and can be implemented in RDMA for such a protocol.

In RCC, CALVIN works as follows. For each epoch, each machine node receives a set of transactions. The sequencing layer in each machine determines the order of the locally received transactions and broadcasts them to all other machines. After the transaction dispatch, each machine has the whole set of transactions in the epoch with a consensus and deterministic order. The transaction dispatch incurs CALVIN 's first source of communication: the transaction inputs, its RS and WS, will be delivered to all other machines. With RPC, such information can be sent in batch and the receiver nodes will store the data locally. With one-sided primitives, the implementation is more challenging, since the sender node needs to be aware of the location to write to remote nodes. We design a specific buffer structure, in each node that is known among all machines, so that the sender can directly use doorbell batched RDMA `WRITE`s to deliver the transaction information to all other nodes and update metadata. The procedure is shown in Figure 9.

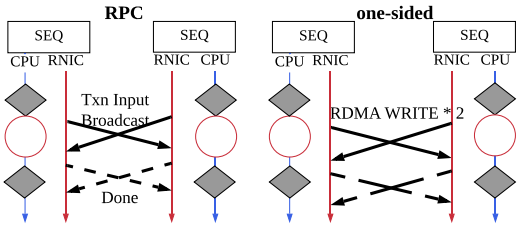


Fig. 9. CALVIN : Txn Input Broadcasting

Figure 10 exemplifies the buffer organization design for CALVIN . RCC CALVIN uses two memory buffers that enable RDMA remote access. 1. CALVIN Request Buffer (CRB). Each CRB contains one CALVIN Header (CH) and a list of CALVIN Requests (CR). Each CH has control information for CALVIN 's scheduler to decide whether it has collected all transaction inputs in one epoch and whether all transactions in a batch have finished execution and all threads should move on to the next epoch. 2. CALVIN Forward Buffer (CFB). Each execution co-routine uses one CFB to receive forwarded values from other machines. We will discuss CALVIN 's value forwarding in later paragraphs. Besides these two buffers

1. The condition `commit_tts` must be greater than `wts` of the record in RS based on how it is updated

above, to support asynchronous replication, each backup machine has a list of CRBs for receiving asynchronous backup requests for each epoch.

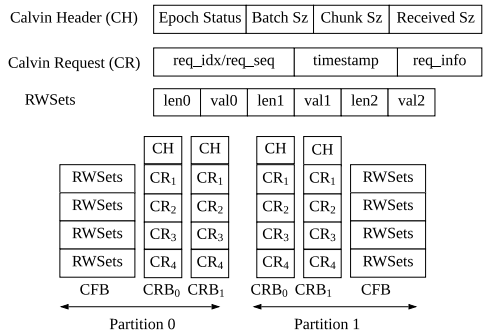


Fig. 10. RDMA-enabled buffer organization with batch size per epoch = 4 and the maximum number of read/write sets per transaction = 3.

CALVIN has the unique execution model that each transaction is executed by multiple machines. Specifically, the machines that have records in WS according to data partition will execute the operations of the transaction that will write these records. These machines are called active participants. The machines that have records in RS are called passive participants, since they do not contain records in WS, they do not execute the transaction but only provide data to active participants. To start the execution in active participants, they need to get all records in RS/WS. This leads to the second communication in CALVIN .

First, the passive participants need to send the local records in RS to all active participants. Second, the active participants need to send the local records in WS to the other active participants. Actively participants will wait and collect all the needed records forwarded from other machines. Two-sided implementations is easier since we can simply use a data structure for holding the mappings from tuple key to their values in the epoch. The one-sided version needs two doorbell-batched RDMA `WRITE`s to forward value and notify the receiver. After the communication, the transactions can execute in active participants.

We only described the key operations in CALVIN that is relevant to communications and omit many details, which can be found in [11]. The main challenge of implementing CALVIN is to design the sophisticated data structures to facilitate the correct communication between machines, especially for one-sided primitives. We choose not to discuss them in detail since it is mainly engineering efforts. Compared to other five protocols, we do not need to consider many subtle issues to ensure correctness, because after transaction dispatch and RS/WS preparation, the execution is mostly local. We believe including CALVIN in RCC is important because we can understand the communication implementation and cost for the shared-nothing protocol. As far as we know, it is also the first implementation of CALVIN with RDMA.

Key RCC insight: We can implement the input broadcasting and value forwarding by two doorbell-batched WRITEs in the one-sided version with a careful design and handling of RDMA-enabled buffer data structures.

5 Hybrid Protocols

RCC can evaluate all protocol stages, a natural question is: what would be the best implementation if we can use different primitives for different stages? DrTM+H [20] only provides the answer for OCC, but what about others?

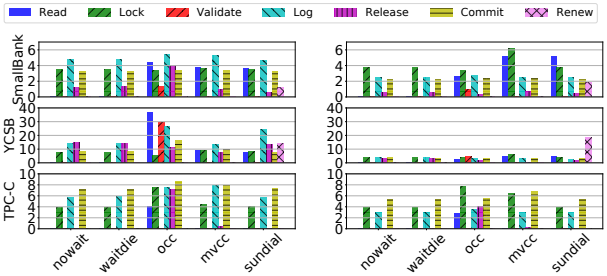


Fig. 11. Latency breakdown: **RPC** vs. **one-sided**

5.1 Methodology

RCC uses two methods for exploiting the potential of hybrid protocols. The first method is based on the stage-wise latency breakdown produced by RCC. Accordingly, the hybrid designs for protocols can be straight-forward by *cherry-picking* the better communication type among the two-sided and one-sided world for each operation. Figure 11 shows the latency-breakdown of all five protocols in RCC using one co-routine under various workloads. As one example, we can see that for SmallBank: 1. a hybrid design of MVCC which includes RPC Read & Lock and one-sided Log & Release & Commit can be a good candidate of hybrid MVCC; 2. a hybrid design of SUNDIAL which includes RPC Read & Renew and one-sided Lock & Log & Commit will incur shorter latency and thus may improve its throughput on SmallBank. With the analysis of latency results, we see that: 1. Log, Commit and Release operations prefer one-sided operations; 2. SUNDIAL's renew operation prefers two-sided RPC; 3. For complex read/lock operation as in MVCC and SUNDIAL, two-sided RPC may be rewarding; and 4. The best hybrid designs of any protocol are workload-dependent.

Alternatively, RCC has implemented all protocols in a way that makes it possible to conduct the exhaustive search of all combinations of hybrid protocols. This is useful when multiple co-routines are involved or when the system load is high. RCC provides a configurable framework that could comprehensively evaluate *any* two-sided, one-sided, and *any* combination of hybrid implementations of protocols included. To achieve this goal, we provide a "code" for each hybrid implementation, each binary digit in the code specifying the primitive to use for each stage. This interface allows RCC to be friendly to both common and expert users: common users can find the best hybrid implementation given the protocol and the workload specification. Expert users can specify their own hybrid code to indicate the primitive used in each stage and verify their intuitions quickly. By leveraging RCC, we aim to find solid evidence of the best hybrid design instead of allowing users to guess and try based on suggestive guidelines. Figure 12 shows a comparison of stage-wise hybrid protocols compared to their purely two-sided RPC or one-sided implementation when 10 co-routines are used for four protocols. Each blue dot in Figure 12 corresponds to a hybrid design with one combination of primitives in different stages. It can be seen that most hybrid designs span in the middle of purely two-sided and purely one-sided designs. Yet there are hybrid ones that do outperform the better of the two both latency-wise and throughput-wise.

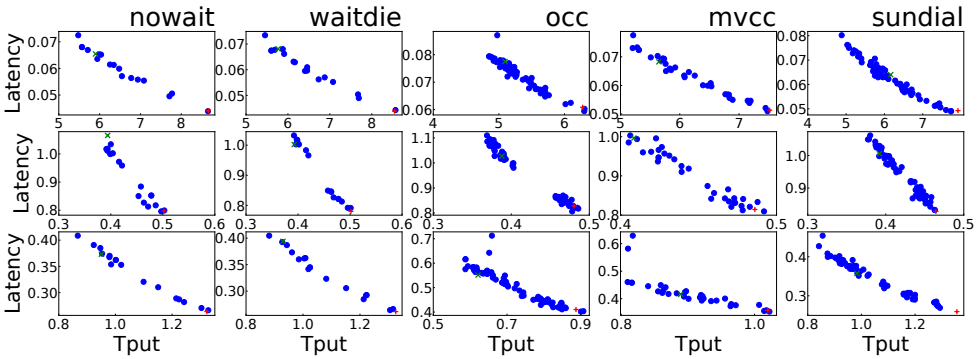


Fig. 12. Performance of all stage-wise hybrid implementations compared with two-sided or one-sided implementation: **RPC** (Green), **one-sided** (Red), **hybrid** (Blue) for three workloads from up to down: SmallBank, YCSB, TPC-C.

5.2 Implementation Requirements

The design of a universal hybrid implementation generator needs to satisfy several requirements to ensure correctness. First, the remote tuple address must be recorded for RPC Read or Lock since future one-sided stages may need the offset to access the tuple. Second, any two-sided or one-sided stage must work correctly, assuming that it may work with another stage using a different primitive. We rely on a shared RDMA-enabled memory region for every tuple in the read/write set to maintain the correct communication between stages. Third, the heterogeneous stages must reach consensus to indicate if one has finished its work correctly. This may cause trickiness if not handled carefully: RPC handler must notify lock requesters of the completion of the lock by not only sending back a success reply, but also writing a success bit in the agreed region in the RDMA-enable memory of the locked tuple so that one-sided Release stage can successfully release the lock.

6 Evaluation

6.1 Workloads and Setups

We use three popular OLTP benchmarks, SmallBank [33], YCSB [35], and TPC-C [34], to evaluate protocols using two-sided *RPC*, *one-sided* primitives, and stage-wise *hybrid* primitives. For each protocol and benchmark, we leverage the cherry-picking methodology described in section 5 to find the best hybrid design, which can happen to be purely one-sided across all stages. We include the traditional TCP-based protocols in the evaluation section. For all benchmarks, records are partitioned across nodes.

SmallBank [33] is a banking application. Each transaction performs reads and writes on the account data. SmallBank features a small number of writes and reads in one transaction (< 3) with simple arithmetic operations, making SmallBank a network-intensive application.

YCSB [35] (The Yahoo! Cloud Serving Benchmark) is designed to evaluate large-scale Internet applications. There is just one table in the database. YCSB parameters such as record size, the number of writes or reads involved in a transaction, the ratio of read/write, the contention level, and time spent at the computation phase are all configurable. In all our experiments, the record length is set to 64 bytes. The number of records in the YCSB table is proportional to the cluster size and the number of transaction threads used. By default, each transaction contains 10 operations: 20% write, and 80% read, and it spends 5 microseconds in its execution phase. The hot area accounts for 0.1% of total records. The contention in YCSB is controlled by allowing a configurable percentage of read/write to access the hot area, which we call the Hot Access Probability, which is 10% probability by default. In Section 6.4, we study the effects of contention.

TPC-C [34] simulates the processing of warehouse orders and is representative of CPU-intensive workloads. In our evaluation, we run the **new-order** transaction since other transactions primarily focus on local operations. The **new-order** accounts for 45% in TPC-C and consists of longer (up to 15) distributed writes and complex transactions.

We evaluate RCC on four nodes of an RDMA-capable EDR cluster, each node equipped with two 12-core Intel Xeon E5-2670 v3 processor, 128GB RAM, and one ConnectX-4 EDR 100Gb/s InfiniBand MT27700. With one RNIC on each node, we run evaluations on the CPU on the same NUMA node with the RNIC to prevent NUMA from affecting our results. By default, we use ten transaction execution threads affiliating to ten cores (12 threads excluding one stat thread and one QP thread) and use 1 co-routine in section 6.2 and 10 co-routines in section 6.4, 6.5, and 6.6. We enable 3-way replication for RCC. The implementations in RCC are evaluated on three metrics: *throughput, latency and abort rate*.

RCC ensures unbiased cross-protocol comparison by enforcing that 1) all protocol implementations share the same RDMA library and thus share the same set of RDMA-related parameters such as max package size and max receive queue size. 2) other system-wide parameters such as co-routines numbers, thread numbers are the same among protocol comparisons. 3) the same set of workload-related parameters are applied among comparisons. These three factors allow only the protocol themselves as the variable, thus making the comparison unbiased.

6.2 Overall Results

Figure 13 shows the results of all three implementations of the six protocols. The results show the effects of different implementations and cross-protocol comparisons.

For the same protocol, the performance of one-side is generally better than RPC, except MVCC under TPC-C. MVCC does not benefit from one-sided primitives on TPC-C, both latency-wise and throughput-wise. As TPC-C contains long 100% write operations, all protocols incur over 50% abort rate. Therefore latency is determined by how quickly an abort decision can be made. one-sided MVCC does not outperform RPC in this scenario since a one-sided MVCC transaction may need two round trips to decide to abort.

Across all one-sided implementations, OCC is one best choice for YCSB, yet it becomes the second-to-the-worst for SmallBank. In fact, one-sided 2PL has better performance

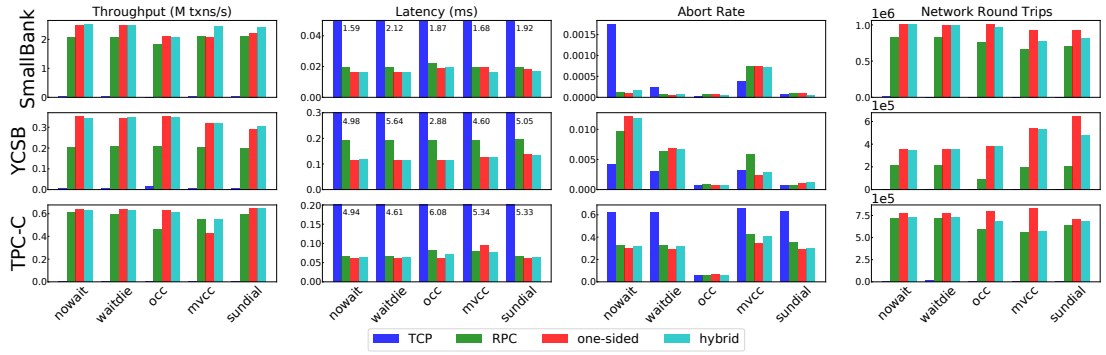


Fig. 13. Overall throughput, latency, abort rate and # of network round trips

on SmallBank over one-sided OCC, MVCC and SUNDIAL. Besides, the best protocol choice depends not only on workload characteristics but also on communication types. For YCSB, the performance of RPC implementations are similar across protocols while one-sided ones peaks at OCC.

Stage	H-MVCC ⋆	H-SUNDIAL ⋆	H-SUNDIAL †
Read	R	R	O
Lock	R	R	O
Validate	N/A	N/A	N/A
Log	O	O	O
Release	O	O	O
Commit	O	O	O
Renew	N/A	R	R

TABLE 1
Hybrid strategy for the three cases that achieve better performance compared to their RPC and one-sided counterparts.(R for RPC and O for one-sided. ⋆ indicates SmallBank and † indicates YCSB.)

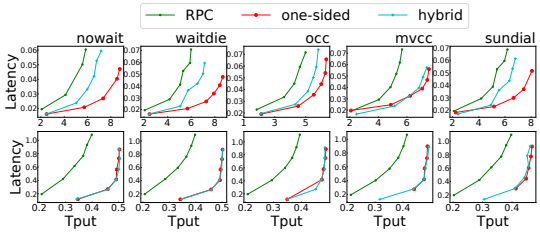


Fig. 14. Throughput (M txns/s) and Latency (ms) for **SmallBank** (Up) and **YCSB** (Down).

6.3 Effect of Co-routines

As for the chosen hybrid implementations, they generally have higher throughput than RPC or one-sided counterpart by 32.2% and up to 67%; we found three hybrid implementations performing better than both throughput-wise. On SmallBank, the hybrid MVCC performs 17.8% and 21.7% better than the RPC and one-sided implementations; the hybrid SUNDIAL performs 14.8% and 8.6% better than its RPC and one-sided counterparts. On YCSB, the hybrid SUNDIAL performs 51.6% and 4.5% better than the RPC and one-sided implementations. Table 1 lists the hybrid strategies for these three occurrences. It is clear that the best hybrid choice for SUNDIAL is workload-dependent.

Figure 14 shows the latency and throughput change when increasing the number of co-routines from 1 to 11 with a step of 2 for both SmallBank and YCSB. We see that the latency is always increased with more co-routines due to the fact that with more co-routines the waiting time of each of them being served in a round table by their serving thread is increased. Also, the throughput increases since more co-routines can hide the latency of network operations. However, we also observe that throughput starts to plateau after a certain number of co-routines. This is due to the higher contention with longer latency. The performance of hybrid implementations lies in the middle between RPC and one-sided ones for SmallBank and similar to the one-sided implementations for YCSB as more co-routines are used. Note that in Figure 14, hybrid designs may not perform equivalently or better with more co-routines since these best hybrid designs are statically determined when #co-routines is 1. With every #co-routines greater than 1, the best hybrid design needs to be re-evaluated by enumeration.

Figure 15 shows the results for CALVIN . Due to its shared-nothing architecture, it is not directly comparable to others. In both RPC and one-sided, we see that increasing #co-routines may or may not improve throughput. This is because CALVIN requires RDMA-based epoch synchronization among all sequencer co-routines on all machines; therefore network latency due to staggered co-routines cannot be hidden with the use of increasing #co-routines. Note that RDMA-based CALVIN does not reach as higher throughput as other protocols compared to their TCP counterpart due to high synchronization cost.

6.4 Effect of Contention Level

Figure 16 shows the throughput of RPC and one-sided implementation of different protocols with different contention levels using YCSB. We control the contention levels by limiting the number of hot records to 0.1% of total records and varying the possibility of read/write visiting hot records.

We have several key observations. With low contention, the throughput differences are small, and the worst one-sided is better than the best RPC. As the contention increases, the thoughput of all protocols decrease, but OCC always drops most significantly because of a larger possibility to abort and high abort cost due to its optimistic assumption under a high contention level. The performance of NOWAIT and WAITDIE also decrease considerably due to the intensive conflict read and write locks. MVCC and SUNDIAL are less affected when the conflict rate increases. As a result, with high contention, the throughput of different protocols become quite different, but the gaps between RPC and one-sided are much smaller. We also notice that one-sided SUNDIAL and MVCC, although featuring advanced read-write conflict management, are worse than one-sided OCC at low contention. It is because these two have more complicated and costly operations to maintain more information to

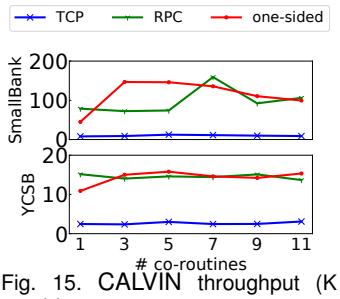


Fig. 15. CALVIN throughput (K txns/s)

reduce the abort rate. After all, every access to remote data will trigger network operation in their one-sided versions. A key conclusion is that OCC is not the best—in fact always the worst with high contention, it can only be justified with a common framework.

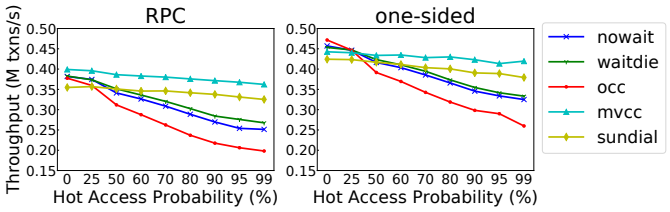


Fig. 16. Effect of contention level on throughput

6.5 Effect of Computation

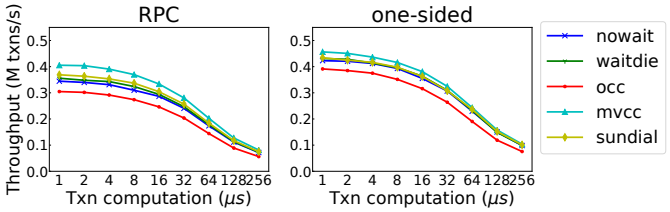


Fig. 17. Effect of computation on throughput

To study the effect of different computation time in the whole life of transaction execution, we add dummy computation in the execution stage of YCSB, ranging from 1 to 256 μs. We show results in Figure 17. We observe that (1) RPC and one-sided share a similar decreasing trend as computation increases; and (2) the advantage of one-sided over RPC is diminishing as the computation increases. For RPC, more computation adds latency to handle RPC request; for one-sided, more computation narrows its advantage over RPC due to the non-involvement of CPU in communication.

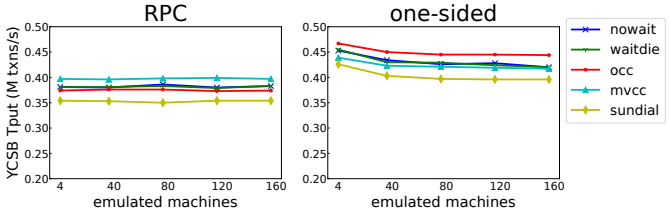


Fig. 18. Throughput on emulated large EDR clusters.

6.6 Scalability of QPs

To understand how protocols perform with a much larger cluster, we run all protocols against the YCSB benchmark (90% hot access probability and 0.1% hot area) on several emulated larger EDR clusters. as shown in Figure 18. Each RDMA operation selects the sending QP from multiple same-destination QPs in a round-robin manner to emulate the network traffic on large clusters. We observe that on emulated larger clusters, one-sided implementations maintain their superiority over RPC ones up to a 160-node cluster, yet the advantage gap closes as cluster size increases. We attribute this behavior to the fact that an increasing number of QPs needed for larger clusters will cause performance loss due to limited NIC capabilities.

7 Related Work

Distributed transaction processing requires dataset partitioned into multiple machines [40]. Distributed data is also common in other areas such as WBAN [41] and cloud systems [42]. Deneva [30] is the most recent work comparing distributed concurrency control protocols in a single unified framework, however all protocols in Deneva are based on TCP. In terms of comparing RDMA primitives, FaRM [19] finds out that RDMA `WRITE`'s polling outperforms `SEND` and `RECV` verbs. [16] shows that UD-based RPC outperforms one-sided primitives. [20] did more primitive-level comparisons with different payload sizes. Compared to them, RCC compares primitives by constructing a wide range of concurrency control algorithms using both primitives. High performance transaction systems have been investigated intensively [8], [9], [11], [17], [18], [20], [26]. Most of them focus on distributed transaction systems [17], [18], [20], [26] since it is more challenging to implement a high performance transaction system with data partitioned across the nodes. Some works, e.g., [9], [16], [17], [18], [20], focus only on one protocol (i.e., some variants of OCC). Other works like [11], [15], [31] explore novel techniques like determinism or leasing. However, these works did not explore the opportunity of using RDMA networks. Compared to NAM-DB [36], which implements a snapshot protocol based on one-sided primitives, RCC builds serializable protocols with both primitives. Besides RCC, many domains, such as machine learning, use the hybrid method [43] to obtain better performance compared to the pure way.

8 Conclusion

We develop RCC, the first unified and comprehensive RDMA-enabled distributed transaction processing framework containing six serializable concurrency control protocols. Our goal is to unbiasedly compare protocols on OLTP workloads in a common execution environment with the concurrency control protocol being the only changeable component. Based on RCC, we get the insights that cannot be obtained by any existing systems. Most importantly, we obtained and analyzed stage-wise latency breakdowns to develop efficient hybrid implementations. Moreover, RCC can enumerate all hybrid implementations of a protocol under a given workload characteristic. RCC is a significant advance over the state-of-the-art; it can provide performance insights and be used as the common infrastructure for fast prototyping new hybrid implementations.

References

[1] C. Curino, E. Jones, Y. Zhang, and S. Madden, "Schism: a workload-driven approach to database replication and partitioning," *Proceedings of the VLDB Endowment*, vol. 3, no. 1-2, pp. 48–57, 2010.

[2] A. Pavlo, C. Curino, and S. Zdonik, "Skew-aware automatic database partitioning in shared-nothing, parallel OLTP systems," in *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. ACM, 2012, pp. 61–72.

[3] R. Agrawal, M. J. Carey, and M. Livny, "Concurrency control performance modeling: Alternatives and implications," *ACM Transactions on Database Systems (TODS)*, vol. 12, no. 4, pp. 609–654, 1987.

[4] P. A. Bernstein, P. A. Bernstein, and N. Goodman, "Concurrency control in distributed database systems," *ACM Comput. Surv.*, vol. 13, no. 2, pp. 185–221, Jun. 1981. [Online]. Available: http://doi.acm.org.libproxy1.usc.edu/10.1145/356842.356846

[5] P. A. Bernstein and N. Goodman, "Multiversion concurrency control—theory and algorithms," *ACM Transactions on Database Systems (TODS)*, vol. 8, no. 4, pp. 465–483, 1983.

[6] H.-T. Kung and J. T. Robinson, "On optimistic methods for concurrency control," *ACM Transactions on Database Systems (TODS)*, vol. 6, no. 2, pp. 213–226, 1981.

[7] H. A. Mahmoud, V. Arora, F. Nawab, D. Agrawal, and A. El Abbadi, "Maat: Effective and scalable coordination of distributed transactions in the cloud," *Proceedings of the VLDB Endowment*, vol. 7, no. 5, pp. 329–340, 2014.

[8] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden, "Speedy transactions in multicore in-memory databases," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM, 2013, pp. 18–32.

[9] C. Lee, S. J. Park, A. Kejriwal, S. Matsushita, and J. Ousterhout, "Implementing linearizability at large scale and low latency," in *Proceedings of the 25th Symposium on Operating Systems Principles*. ACM, 2015, pp. 71–86.

[10] M. Stonebraker, "The case for shared nothing," *IEEE Database Eng. Bull.*, vol. 9, no. 1, pp. 4–9, 1986.

[11] A. Thomson, T. Diamond, S.-C. Weng, K. Ren, P. Shao, and D. J. Abadi, "Calvin: Fast distributed transactions for partitioned database systems," in *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '12. New York, NY, USA: ACM, 2012, pp. 1–12. [Online]. Available: http://doi.acm.org/10.1145/2213836.2213838

[12] P. Bailis, A. Fekete, M. J. Franklin, A. Ghodsi, J. M. Hellerstein, and I. Stoica, "Coordination avoidance in database systems," *Proceedings of the VLDB Endowment*, vol. 8, no. 3, pp. 185–196, 2014.

[13] Y. Jiang, W. Liu, X. Shi, and W. Qiang, "Optimizing the copy-on-write mechanism of docker by dynamic prefetching," *Tsinghua Science and Technology*, vol. 26, no. 3, pp. 266–274, 2021.

[14] A. Kalia, M. Kaminsky, and D. G. Andersen, "Using rdma efficiently for key-value services," in *Proceedings of the 2014 ACM Conference on SIGCOMM*, ser. SIGCOMM '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 295–306. [Online]. Available: https://doi.org/10.1145/2619239.2626299

[15] X. Wei, J. Shi, Y. Chen, R. Chen, and H. Chen, "Fast in-memory transaction processing using RDMA and HTM," in *Proceedings of the 25th Symposium on Operating Systems Principles*. ACM, 2015, pp. 87–104.

[16] A. Kalia, M. Kaminsky, and D. G. Andersen, "Fasst: Fast, scalable and simple distributed transactions with two-sided RDMA datagram RPCs," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, 2016, pp. 185–201.

[17] Y. Chen, X. Wei, J. Shi, R. Chen, and H. Chen, "Fast and general distributed transactions using RDMA and HTM," in *Proceedings of the Eleventh European Conference on Computer Systems*. ACM, 2016, p. 26.

[18] A. Dragojević, D. Narayanan, E. B. Nightingale, M. Renzelmann, A. Shamis, A. Badam, and M. Castro, "No compromises: distributed transactions with consistency, availability, and performance," in *Proceedings of the 25th Symposium on Operating Systems Principles*. ACM, 2015, pp. 54–70.

[19] A. Dragojević, D. Narayanan, M. Castro, and O. Hodson, "FaRM: Fast remote memory," in *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, 2014, pp. 401–414.

[20] X. Wei, Z. Dong, R. Chen, and H. Chen, "Deconstructing RDMA-enabled distributed transactions: Hybrid is better!" in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, 2018, pp. 233–251.

[21] S.-Y. Tsai and Y. Zhang, "Lite Kernel RDMA support for Datacenter Applications," in *Proceedings of the 26th Symposium on Operating Systems Principles*. ACM, 2017, pp. 306–324.

[22] S. Mu, Y. Cui, Y. Zhang, W. Lloyd, and J. Li, "Extracting more concurrency from distributed transactions," in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, 2014, pp. 479–494.

[23] S. Roy, L. Kot, G. Bender, B. Ding, H. Hojjat, C. Koch, N. Foster, and J. Gehrke, "The homeostasis protocol: Avoiding transaction coordination through program analysis," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. ACM, 2015, pp. 1311–1326.

[24] R. Escriva, B. Wong, and E. G. Sirer, "Warp: Lightweight multi-key transactions for key-value stores," *arXiv preprint arXiv:1509.07815*, 2015.

[25] J. Cowling and B. Liskov, "Granola: low-overhead distributed transaction coordination," in *Presented as part of the 2012 USENIX Annual Technical Conference (USENIX ATC 12)*, 2012, pp. 223–235.

[26] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild *et al.*, "Spanner: Google's globally distributed database," *ACM Transactions on Computer Systems (TOCS)*, vol. 31, no. 3, p. 8, 2013.

[27] J. M. Faleiro and D. J. Abadi, "Rethinking serializable multiversion concurrency control," *arXiv preprint arXiv:1412.2324*, 2014.

[28] X. Yu, "An evaluation of concurrency control with one thousand cores," Ph.D. dissertation, Massachusetts Institute of Technology, 2015.

[29] X. Yu, G. Bezerra, A. Pavlo, S. Devadas, and M. Stonebraker, "Staring into the abyss: An evaluation of concurrency control with one thousand cores," *Proceedings of the VLDB Endowment*, vol. 8, no. 3, pp. 209–220, 2014.

[30] R. Harding, D. Van Aken, A. Pavlo, and M. Stonebraker, "An evaluation of distributed concurrency control," *Proceedings of the VLDB Endowment*, vol. 10, no. 5, pp. 553–564, 2017.

[31] X. Yu, Y. Xia, A. Pavlo, D. Sanchez, L. Rudolph, and S. Devadas, "Sundial: harmonizing concurrency control and caching in a distributed OLTP database management system," *Proceedings of the VLDB Endowment*, vol. 11, no. 10, pp. 1289–1302, 2018.

[32] A. Kalia, M. Kaminsky, and D. G. Andersen, "Design guidelines for high performance RDMA systems," in *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, 2016, pp. 437–450.

[33] T. H.-S. Team, "SmallBank Benchmark," http://hstore.cs.brown.edu/ documentation/deployment/benchmarks/ smallbank/, 2018.

[34] T. T. P. Council, "TPC-C Benchmark V5.11," http://www.tpc.org/tpcc/, 2018.

[35] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with YCSB," in *Proceedings of the 1st ACM symposium on Cloud computing*. ACM, 2010, pp. 143–154.

[36] E. Zamanian, C. Binnig, T. Kraska, and T. Harris, "The end of a myth: Distributed transactions can scale," *arXiv preprint arXiv:1607.00655*, 2016.

[37] J. W. Stamos and F. Cristian, "Coordinator log transaction execution protocol," *Distrib. Parallel Databases*, vol. 1, no. 4, p. 383–408, Oct. 1993. [Online]. Available: https://doi.org/10.1007/BF01264014

[38] D. L. Mills, "A brief history of ntp time: Memoirs of an internet timekeeper," *SIGCOMM Comput. Commun. Rev.*, vol. 33, no. 2, p. 9–21, Apr. 2003. [Online]. Available: https://doi.org/10.1145/956981.956983

[39] K. Lee, J. C. Eidson, H. Weibel, and D. Mohl, "Ieee 1588-standard for a precision clock synchronization protocol for networked measurement and control systems," in *Conference on IEEE*, vol. 1588, 2005, p. 2.

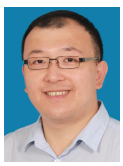
[40] M. S. Mahmud, J. Z. Huang, S. Salloum, T. Z. Emara, and K. Sadatdiynov, "A survey of data partitioning and sampling methods to support big data analysis," *Big Data Mining and Analytics*, vol. 3, no. 2, pp. 85–101, 2020.

[41] X. Tan, J. Zhang, Y. Zhang, Z. Qin, Y. Ding, and X. Wang, "A puf-based and cloud-assisted lightweight authentication for multi-hop body area network," *Tsinghua Science and Technology*, vol. 26, no. 1, pp. 36–47, 2021.

[42] W. Zhang, X. Chen, and J. Jiang, "A multi-objective optimization method of initial virtual machine fault-tolerant placement for star topological data centers of cloud systems," *Tsinghua Science and Technology*, vol. 26, no. 1, pp. 95–111, 2021.

[43] G. Pu, L. Wang, J. Shen, and F. Dong, "A hybrid unsupervised clustering-based anomaly detection method," *Tsinghua Science and Technology*, vol. 26, no. 2, pp. 146–153, 2021.

Chao Wang Chao Wang is a Ph.D. student at University of Southern California, working at the ALCHEM lab. His major research interest is in parallel and distributed systems, ML compilers and systems.



Xuehai Qian Xuehai Qian is an assistnat professor at University of Southern California. His research interests include domain-specific systems and architectures, performance tuning and resource management of cloud systems, and parallel computer architectures. He got his Ph.D from University of Illinois Urbana Champaign and was a postdoc at UC Berkeley. He is the recipient of W.J Poppelbaum Memorial Award at UIUC, NSF CRII and CAREER Award, and the inaugural ACSIC (American Chinese Scholar In Computing) Rising Star Award.