A Comprehensive Evaluation of RDMA-enabled Concurrency Control Protocols

Chao Wang
University of Southern California
wang484@usc.edu

Kezhao Huang
Tsinghua University
huang-kz16@mails.-tsinghua.edu.cn

Xuehai Qian
University of Southern California
xuehai.qian@usc.edu

ABSTRACT

On-line transaction processing (OLTP) applications require efficient distributed transaction execution. When a transaction accesses multiple records in remote machines, network performance is a crucial factor affecting transaction latency and throughput. Due to its high bandwidth and very low latency, RDMA (Remote Direct Memory Access) has achieved much higher performance for distributed transactions than traditional TCP-based systems. RDMA provides primitives for both two-sided and one-sided communication. Although recent works have intensively studied the benefits of RDMA in distributed transaction systems, they either focus on primitive-level comparisons of two communication models (one-sided vs. two-sided) or only study one concurrency control protocol. A comprehensive understanding of the implication of RDMA for various concurrency control protocols is an open problem.

In this paper, we build *RCC*, the first unified and comprehensive RDMA-enabled distributed transaction processing framework supporting six concurrency control protocols using either two-sided or one-sided primitives. We intensively optimize the performance of each protocol without bias, using known techniques such as co-routines, outstanding requests, and doorbell batching. Based on RCC, we conduct the first and most comprehensive (to the best of our knowledge) study of the six representative distributed concurrency control protocols on two clusters with different RDMA network capabilities.

PVLDB Reference Format:
Chao Wang, Kezhao Huang, Xuehai Qian. A Comprehensive Evaluation of RDMA-enabled Concurrency Control Protocols. *PVLDB*, 12(xxx): xxxx-yyyy, 2019.
DOI: https://doi.org/10.14778/xxxxxxx.xxxxxxx

1. INTRODUCTION

On-line transaction processing (OLTP) has ubiquitous applications in important domains, including banking, stock marketing, e-commerce, etc. As the exponential growth of data volume, single-server Database Management Systems (DBMS) are experiencing extreme difficulties in handling a large number of queries from clients due to limited system resources. Partitioning data sets across distributed machines is necessary and becoming increasingly important. However, partitioning data such that all queries access only one partition is challenging [10, 24]. Therefore, transaction executions inevitably access a set of networked machines.

Distributed transactions should guarantee (1) 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, researchers have investigated distributed transactions for decades. Prior works have proposed many distributed concurrency control protocols such as two-phase Locking (2PL) [3], timestamp-based [3], multi-versioned [4], optimistic protocols (OCC)-[20], etc.

The well-known challenge of multi-partition serializable concurrency control protocols is the significant performance penalties[26] [28]. When a transaction accesses multiple records over the network, any other transactions it conflicts with have to be serialized [2]. Therefore, a high-speed network plays a crucial role in a distributed DBMS system.

Remote Direct Memory Access (RDMA) is a new technology that enables the network interface card (NIC) to access the memory of a remote server in a distributed cluster. Due to its high bandwidth and very low latency, RDMA has been actively used in distributed transaction systems [32, 18, 5, 12] and has enhanced the performance by orders of magnitude compared to traditional systems using TCP/IP.

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 even remote CPUs. There have already been exhaustive studies investigating the pros and cons of using each primitive to understand their performance implications. Recently, several works compared two-sided primitives versus one-sided ones [16, 11, 12, 31, 29] at *primitive-level* using micro-benchmarks.

Unfortunately, primitive-level comparisons do not directly transfer to insights of building high-performance transaction processing systems. How to effectively leverage the two types of primitives is still a difficult consideration. Two takeaways from recent work RTX [31] are: (1) the best performance of a specific concurrency control protocol (i.e., OCC) cannot be simply achieved by solely using one-sided or two-sided communication; and (2) different communication

Proceedings of the VLDB Endowment, Vol. 12, No. xxx
ISSN 2150-8097.
DOI: https://doi.org/10.14778/xxxxxxx.xxxxxxx

primitives are suitable for different protocol *stages*. These conclusions suggest that achieving the best performance of a concurrency control protocol using RDMA technology is far from trivial and demands a systematic investigation.

Recent works such as [18, 11, 31] all focus on the design and optimization of OCC using RDMA. However, with a great number of other well-known and vital concurrency control protocols [23, 22, 25, 13, 9], the best implementation practices using RDMA and its implication on *different protocols* are still unknown. Specifically, we ask several key **open questions**: (1) For each protocol, *how* to leverage RDMA to construct an efficient implementation? (2) For each protocol, between two-sided and one-sided communication, *which* choice can achieve better performance? (3) In the context of RDMA, how can we perform *apple-to-apple* comparison among the performance and characteristics of *different* protocols? (4) For different protocols, what are the implications of RDMA network performance? Are the implications similar or largely different? What is the relative performance of different protocols with RDMA compared to TCP [14]? This list of questions continues. Unfortunately, current research on RDMA-enabled distributed transaction systems is insufficient to answer these critical questions.

In this paper, we take the first step to provide definitive answers to the above questions. We build *RCC*, the first unified and comprehensive RDMA-enabled distributed transaction processing framework supporting *six* concurrency control protocols, including NOWAIT [3], WAITDIE [3], OCC [20], MVCC [4], SUNDIAL [35], and CALVIN [28]. Based on RCC, we conduct the *first and most comprehensive* (to the best of our knowledge) comparison among *different* representative distributed concurrency control protocols in the context of RDMA. The wide range of protocols supported in RCC includes: (1) classical protocols that have been used in production DBMS for decades such as 2PL (NOWAIT and WAITDIE); (2) OCC, which has attracted most recent research interests; (3) MVCC and SUNDIAL, which use timestamp to allow more concurrency; and (4) CALVIN, a recently proposed protocol to ensure deterministic transaction execution. We believe that the diverse set of protocols can well represent recent research trends and industry practices of concurrency control algorithms. It is imperative to understand the potential and implication of RDMA technology in this context.

In RCC, we provide two implementations for each protocol: using *two-sided* RDMA-enabled Remote Procedure Call (RPC) and one-sided communication primitives. We intensively optimize the performance without bias using known techniques such as co-routines [18], outstanding requests [31] and doorbell batching [17]. We evaluate all protocols in RCC on *two clusters* with different RDMA network capabilities. One is equipped with ConnectX-4 EDR 100Gb/s InfiniBand (EDR) and the other with ConnectX-3 Pro FDR 56Gb/s InfiniBand (FDR). We use them due to the huge difference in supporting RDMA one-sided operations: the latency of a single one-sided operation using FDR InfiniBand can be 5x more than that of using the EDR InfiniBand. We evaluate RCC on three OLTP workloads: SmallBank [27], TPC-C [8], and YCSB [6].

Compared to RTX [31], which only studied OCC, RCC covers significantly more representative protocols, providing the opportunity to perform a much more comprehensive study and revealing more insights. Compared to Deneva [14] which is the latest in-memory distributed database evaluation framework also including six (slightly different) concurrency control protocols, RCC implements each protocol using RDMA communication primitives instead of TCP/IP, providing more guidelines and insights for building RDMA-friendly transaction systems.

2. BACKGROUND

2.1 RDMA and Its Primitives

RDMA (Remote Direct Memory Access) is a network technology featuring high bandwidth and low latency data transfer with low CPU overhead. It is widely considered suitable for large data centers. RDMA operations, a.k.a, **verbs**, can be classified into two types: (1) *two-sided* operations including RDMA `SEND/RECV`; and (2) *one-sided* operations including RDMA `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/IP.

Queue pairs (QPs) must be set up for RDMA communication. 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 RDMA `RECV` for the corresponding RDMA `SEND` in advance. It polls the receive queue to obtain the data.

QP has three different transport modes: Reliable Connection (RC), Unreliable Connection (UC), and Unreliable Datagram (UD). One RC QP can send messages reliably to exactly one another connected RC QP; a UD QP can either unicast to one or multicast to many UD QPs without delivery guarantees. One-sided operations are only fully supported between two RC QPs, while two-sided operations must be used for UD QPs. 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. On the contrary, a node needs to maintain only one UD QP to send/receive data across the cluster, thus saving system resources.

2.2 RDMA-enabled Distributed Transactions

Distributed transaction systems are popular applications with demanding network operations. Intensive research has been conducted on employing RDMA for distributed transactions. [5] uses pure one-sided operations to transfer and update records. [18] uses UD to implement RPC in its transaction framework. [31] brings up a hybrid of one-sided and two-sided operations for different stages of transactions. All these frameworks focus on some variant of OCC [20].

3. SYSTEM OVERVIEW

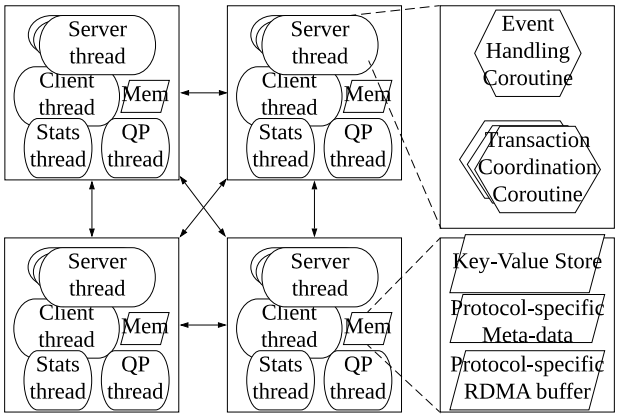

Figure 1: RCC Framework Overview

3.1 Principles and 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 continuously sends transaction requests to a random 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 pairing of RDMA QPs using TCP connections.

RCC uses co-routines as an essential optimization technique [18] for hiding network latency. To this end, 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 releasing of a contending 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. In addition to the in-memory database, each protocol has its protocol-specific metadata to serialize transaction execution. For some protocols like CALVIN, protocol-specific RDMA buffers are required for collecting transaction inputs and forwarding local reads.

3.2 Transaction and Execution Model

RCC employs the 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, the in-memory database keeps all records. Since one-sided operations can only access remote memory by leveraging the pre-computed 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. A *record* refers to the actual data; a *tuple* refers to a record associated with the relevant metadata.

A transaction contains reads and writes of records that are performed atomically. The execution of a transaction contains three stages: 1) *fetching*, where a read/write operation fetches the record and/or metadata if needed; 2) *execution*, where a transaction performs the actual computation locally using the record just fetched; and 3) *commit*, where a transaction checks the metadata to find out if the transaction execution is serializable and updates remote records. Our implementations can be applied to transactions with one or more fetching and execution stages.

3.3 RDMA Primitives and Optimizations

We use two-sided RDMA primitives over UD QP to implement RPC. From [18], two-sided primitives over UD QPs have better performance for symmetric transaction systems than one-sided primitives, and UD mode is much more reliable than expected with RDMA network's lossless link layer. [31] further confirms the unsuitability of one-sided primitives to implement fast RPC compared to two-sided ones.

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 op 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.

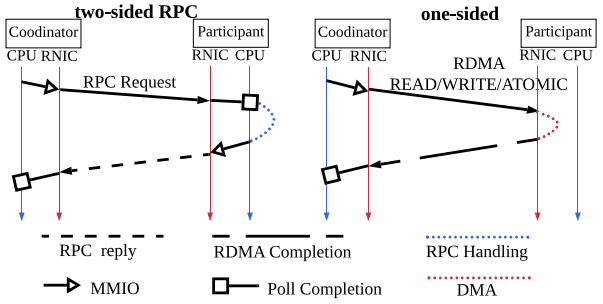

Figure 2: Two-sided RPC versus one-sided

MMIO is an expensive operation to notify RNIC of a request fetching event. Therefore using one MMIO for a batch of RDMA requests can effectively save PCIe bandwidth and improve the performance of transaction systems [31]. Meanwhile, having multiple outstanding requests on the fly can help save the waiting time of request completion, thus reducing the latency of remote transactions [31]. Besides co-routines, RCC uses similar techniques as important optimizations.

4. CONCURRENCY CONTROL

In RCC, we implement six concurrency control protocols with RDMA-enabled two-sided and one-sided communication primitives. The implementations of these protocols

involve a variety of techniques. Among these protocols, NOWAIT [3] and WAITDIE [3] are two examples of 2-phase locking (2PL) [3] concurrency control algorithms. They differ in conflict resolution — how conflicts are handled to ensure serialization. Compared to 2PL, Optimistic Concurrency Control (OCC) [20] reads records speculatively without locking and validates data upon transaction commits — the only time to use locks. MVCC [4] optimizes the performance of read-heavy transactions by allowing the read of stale records instead of aborting. SUNDIAL [35] leverages dynamically changing logical leases to determine the transaction commit orders to reduce aborts. CALVIN [28] introduces determinism to sequence transaction inputs and schedule transactions in a deterministic fashion.

While the protocols themselves are known before, the new contribution of RCC is to rethink their *implementations in the context of RDMA*. For each protocol, we implement two versions: 1) RPC version, which mostly uses remote function call enabled by RDMA's two-sided communication primitives (i.e., `SEND/RECV`); and 2) one-sided version, which mainly uses RDMA's unique one-sided communication primitives (i.e., `WRITE/READ/ATOMIC`). Each version does not solely use one type of RDMA primitives — we choose the proper primitives to implement certain operations when the alternative is overwhelmingly worse.

NOWAIT	lock	record			
WAITDIE	tts	record			
OCC	lock	seq	record		
MVCC	tts	rts	wts[0..3]	record[0..3]	
Sundial	lock	rts	wts	record	
Calvin	lock	record			

Figure 3: Protocol Metadata

4.1 Design Principle

The trade-off between two-sided and one-sided RDMA primitives is well-known. In the RPC version, operations are sent to and executed on the remote machine. The advantage is that the whole operation can be done with one function call, thus saving communication. But the local machine needs to wait for the remote execution, which is typically done by a co-routine after the remote CPU is interrupted. In contrast, one-sided primitives bypass remote CPU and usually achieve better performance for applications with higher CPU usage. However, the local machine may launch more network requests and needs to maintain metadata for remote offsets. Also, one-sided atomic operation tends to be costly. Based on these trade-offs, we consider several common operations used in concurrency control protocols.

Locking. All six protocols need locking. The implementation choice depends on the system load of remote machines executing co-routines. If the remote machine has longer co-routine execution time, the remote lock handler in the RPC version will wait longer, and one-sided implementation is better. We observe this behavior in YCSB in which the co-routine execution time can be configured in the benchmark. For TPC-C and SmallBank, RPC implementation is better since the co-routine execution time is short.

Validation. This operation is needed in OCC, MVCC, and SUNDIAL. RPC version requires only one network operation for validation. But similar to locking, the best choice also depends on the co-routine execution time in remote machines.

Commit. All protocols require it. Except for CALVIN, which only commits locally, one-sided implementation is better since the operation typically needs to write updated data in a remote machine. Moreover, the overhead of accessing metadata can be avoided by caching them in advance.

Next, we describe the step-by-step implementations of five concurrency control protocols except for OCC, which is implemented based on RTX [31]. Figure 4 shows the legend used throughout this section. We try our best to give detailed descriptions so that the paper can also serve as a reference for others to implement the protocols in RDMA. No prior work has shown the comprehensive designs, and we will also open-source our framework if the paper is accepted.

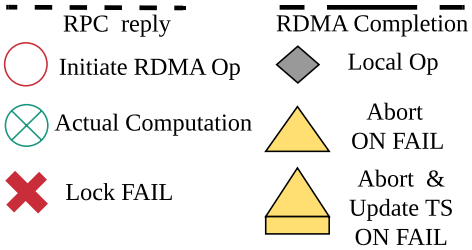

Figure 4: Legend used for this section

4.2 NOWAIT

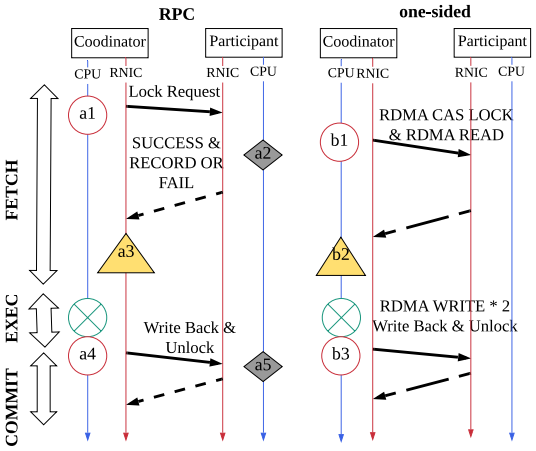

Figure 5: NOWAIT Implementations

NOWAIT [3] 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 former transaction aborts immediately. Figure 5 shows our RPC and one-sided implementations of NOWAIT.

RPC In the data fetch stage, the locking of read/write records is done by sending RPC locking request to the corresponding participant (step **a1**). An RPC handler is called to lock the record using local CAS (step **a2**). If the CAS fails, a failure message is sent back, and the coordinator will release all read and write locks by posting RPC release requests before aborting the transaction (step **a3**). Otherwise, the participant's handler has already found and locked the record locally, and it then returns a success message together with the actual record as an RPC reply (step **a2**). Upon all records are collected, the transaction runs locally at the coordinator side. At the commit stage, a write-back request with the updated record is sent back to each participant (step **a4**), where an RPC handler performs write-back of the record and releases the lock (step **a5**).

One-sided In data fetch stage, RDMA atomic operations are issued by the coordinator to lock remote records

(step **b1**). The transaction aborts if the RDMA `CAS` fails (step **b2**). In the same step **b1**, the coordinator issues an RDMA `READ` immediately after the RDMA `CAS` using the offset of remote record to load the record since the record cannot be changed if the lock succeeds. In case of a failed locking attempt, the returned record is simply ignored. The read offsets are collected and cached by the coordinator before transaction execution starts and thus do not incur much overhead. At the commit stage, two RDMA `WRITEs` are posted to update and unlock the record (step **b3**). Only the second RDMA write is signaled to avoid sending multiple MMIOs and wasting PCIe bandwidth. Such a doorbell batching mechanism, which is also used in RTX [31], provides 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, and we can reduce latency and context switching overhead.

With high contention, lock&read doorbell batching tends to add wasted network traffic. For network-intensive applications with low contention (i.e., SmallBank), the throughput increases by 25.1% while average latency decreases by 22.7% with the above doorbell batching optimization. The lock&read in the data fetch stage and the update&unlock in the commit stage with doorbell batching are used in one-sided implementations of five protocols except for CALVIN.

4.3 WAITDIE

WAITDIE avoids the drawback of NOWAIT— unnecessarily aborting any transactions accessing conflicting records. Instead, it resolves conflicts with a global consensus priority, i.e., a globally unique timestamp. When starting, each transaction obtains a monotonously increasing timestamp. Unlike NOWAIT, where a transaction only tries to log a single bit (i.e., 1) onto a 64-bit record lock upon locking, a transaction in WAITDIE logs its timestamp instead. Upon detecting a conflict, the transaction compares its own timestamp with the logged timestamp to decide whether an abort is needed. Figure 6 shows the data fetch stage of both RPC and one-sided implementations. Other stages are similar to NOWAIT.

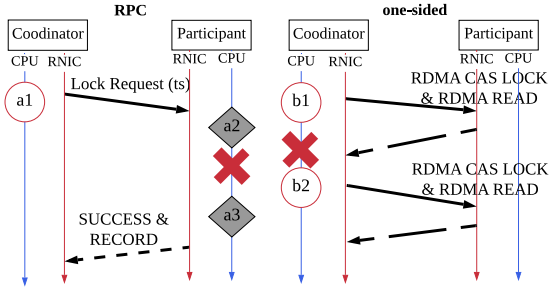

Figure 6: WAITDIE: the FETCH stage

RPC When an accessed record is locked, the lock request handler decides based on the request's timestamp whether to wait on the record until it is unlocked or send back a failure reply immediately (step **a2**). The handler cannot busy wait for the lock and block other incoming requests. Instead, the transaction requesting the lock is added to the lock's waiting list and checked in the event loop periodically by the handler thread (step **a3**). On a lock release, the handler thread removes the transaction from the waiting list and replies to the coordinator with a lock success message and the locked record.

One-sided Similar to NOWAIT, RDMA `CAS` followed by an RDMA `READ` are sent by the coordinator to retrieve the remote lock together with its timestamp and record (step **b1**). Unlike NOWAIT, the timestamp of the current requesting transaction and that of the lock holding transaction are compared to determine whether to abort the transaction or let it wait. With co-routine mechanism, when a transaction decides to wait, it keeps posting RDMA `CAS` with `READ` requests and yields after every unsuccessful trial until it succeeds (step **b2**).

4.4 MVCC

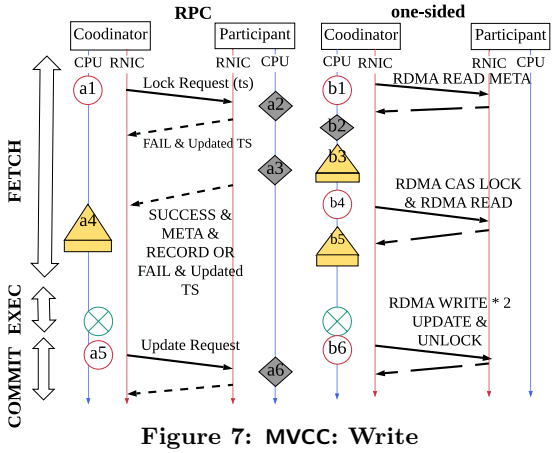

Figure 7: MVCC: Write

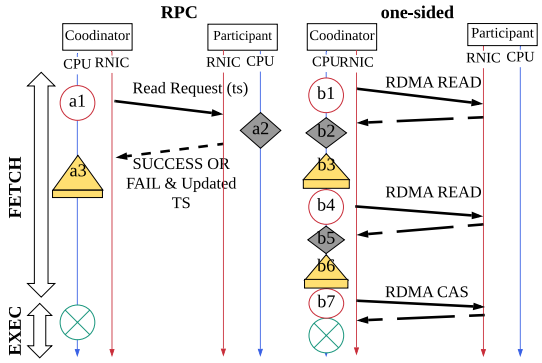

Figure 8: MVCC: Read

MVCC (Multi-Version Concurrent Control) [4] cuts down read-write conflicts by keeping and providing old record versions when possible. For read operations, MVCC looks for proper record versions based on the transaction's current timestamp.

The original MVCC requires using a linked list to maintain a set of record versions. 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, our MVCC implementations use a static number of memory slots (i.e., 4) allocated for each record to store the stale versions. A transaction has to abort when it cannot find a suitable version in the slots for its read operation. Thus, the number of slots determines the trade-off between memory/traversal overhead and read abort rate. We use four slots because our experiments show that at most 4.2% of read aborts are due to slot overflow.

Shown in Figure 3, the metadata of MVCC consists of three parts: 1) write lock, which contains the timestamp of the current transaction holding the lock (`tts`); 2) read timestamp (`rts`), which is the latest (largest) transaction timestamp that has successfully read the record; and 3) write timestamps (`wts`), which are the timestamps of transactions

that have successfully performed writes on the record.

A transaction generates a unique timestamp (`tts`) by appending a unique transaction identifier (e.g., generated by encoding machine ID and co-routine ID) to the low-order bits of a local clock time [4]. The local clock reduces bandwidth overhead of a global clock but may introduce significant bias. While not affecting correctness, the large gap between `wts` on different machines may lead to a long waiting time. To mitigate the issue, each machine adjusts its local timer whenever it finds a larger `wts` or `rts` in any tuple received. This mechanism limits the gap of local timer between machines and reduces the chance of abort due to the lack of suitable record version and the performance impact of the fixed version slots.

RPC The implementation of *write* operations is shown in Figure 7. In (step **a1**), the coordinator sends an RPC request with its `tts` to the participant, which handles it locally in step **a2** and step **a3**. The participant will attempt to lock the record using `tts` (step **a2**) if: 1) the record is not locked, and 2) the transaction's `tts` is larger than the tuple's maximum `wts` *and* its current `rts`. Otherwise, write failure information with the updated timestamp is sent back in response, which potentially updates local timer to limit the gap. On receiving the failure message, the coordinator aborts and updates its `tts` before retry (step **a4**). If the lock attempt was successful, the above condition 2 is re-checked (step **a3**), because writes may happen between condition 1 and the successful locking. After the second check is passed, the participant will eventually traverse `wts` of the record and replying with the latest version of record in step **a3** and a success message. After executing the computation based on received data, in step **a5**, the coordinator sends an RPC request with updated data. Then, the participant writes the data back in step **a6** and releases the lock.

The *read* operation is shown in Figure 8. The coordinator sends the RPC read request with its `tts` to the participant (step **a1**). In step **a2**, the participant traverses all `wts` of the requested record and finds the suitable version for the read. Since there is no read lock in MVCC, the participant needs to traverse the `wts` again after preparing the response data to ensure no writes happened during the reading period. Note that all operations of **a2** are performed locally without a high overhead. On receiving the response, the coordinator will abort or continue based on the message.

One-sided The *write* operation is shown in Figure 7. In step **b1**, the coordinator posts an RDMA `READ` to read the MVCC metadata and the records on the participant. Then, in step **b2**, the received data are checked locally. A write operation can only succeed if the transaction's `tts` is larger than the received `rts` and all `wts` in the tuple (same as in RPC). If the check fails, the coordinator updates its `tts` to the largest timestamp and abort. Otherwise, similar to NOWAIT and WAITDIE, the coordinator posts RDMA `ATOMIC CAS` and RDMA `READ` to lock the remote record and fetches the metadata and record in **b4**. If `ATOMIC CAS` fails, the transaction aborts and updates its `tts` if the timestamp on the lock is larger. In the commit stage, the coordinator posts two RDMA `WRITE`s in step **b6**. The first write updates `wts` and the record, and the second write releases the lock.

Figure 8 shows the *read* operation. In step **b1**, an RDMA `READ` is posted to fetch the metadata and records on the participant. The version check is performed locally in **b2**. Upon failure, the coordinator aborts and updates the `tts` in **b3**. Otherwise, another RDMA `READ` is posted to read the metadata again in **b4**, which is re-checked in **b5** to ensure no writes after the first read (**b1**). Although the data will be ready for execution, we still need to update `rts` on the remote record. In step **b7**, an RDMA `ATOMIC CAS` is posted if this transaction has a larger `tts` than the `rts` on the tuple.

Compared to RPC, the one-sided version generates many more network operations. A network operation is incurred for each access to the remote data. The use of timestamps in MVCC exaggerates this effect.

4.5 SUNDIAL

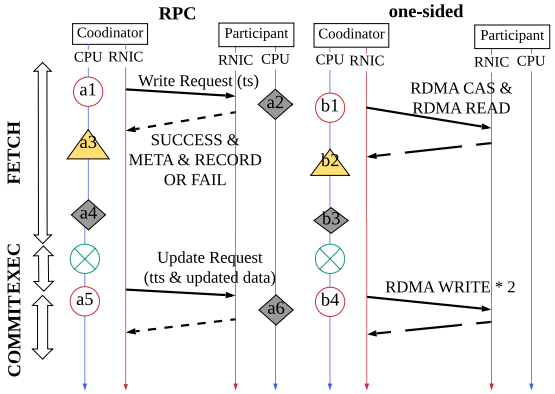

Figure 9: SUNDIAL: Write

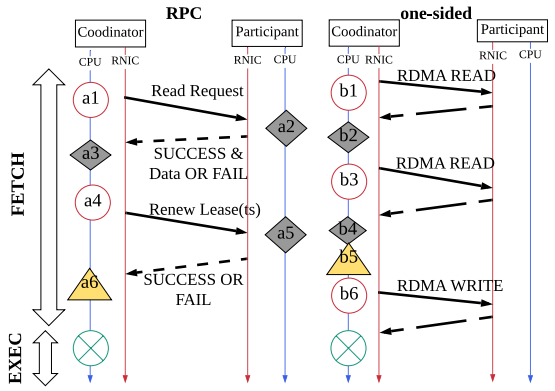

Figure 10: SUNDIAL: Read

SUNDIAL [35] dynamically chooses transaction logical order and is essentially a combination of WAITDIE and OCC. For write conflicts, with WAITDIE, only one transaction can succeed, and others abort early. For read-write conflict, a transaction in SUNDIAL dynamically changes the lease of the tuples and its transaction timestamp, i.e., `tts`. Thus, SUNDIAL can correctly write a record even if another transaction is reading it.

RPC Figure 9 shows the *write* operation. In step **a1**, a coordinator posts a lock RPC request to the participant, who tries to lock the record using local CAS operation in the handler in step **a2**. The coordinator aborts if the handler returns failure. Otherwise, the locked record and metadata are replied. Upon receiving the reply, a local update of `tts` is performed in step **a4**. Then, the coordinator advances `tts` to $rts + 1$. At commit stage, the updated data and the new `wts` (the final `tts`) are sent in an RPC request in step **a5**. In step **a6**, the data and `wts` are updated before the handler releases the lock.

The *read* operation is shown in Figure 10. First, the coordinator posts a request to fetch the record and metadata

(step **a1**). Then, the handler prepares the requested data and re-checks the `wts` before sending (step **a2**). The coordinator then updates `tts` according to the received `wts` according to this equation: $\texttt{tts} = max(\texttt{tts}, wts)$. After fetching the data for all reads and writes, if `tts` is still within the lease of the tuple being read, i.e.,$tuple.wts \leq \texttt{tts} \leq tuple.rts$, the read is valid. Otherwise, in step **a4**, an RPC request is sent to the participant to renew the lease. The transaction aborts if the renewal fails.

One-sided For *write* operations, in step **b1** in Figure 9, the coordinator tries to lock the remote tuple and fetch the metadata and the record using the combined RDMA `ATOMIC CAS` and RDMA `READ` in step **b1**. The coordinator also updates the `tts` using received metadata in step **b3**. In step **b4**, after the computation, the coordinator updates and unlocks the remote record using two RDMA `WRITE`.

For *read* operations, the first step (step **b1** in Figure 10) is to read the whole remote tuple directly using an RDMA `READ`. After fetching the metadata, local operations in step **b2** is performed to update `tts`. Then the coordinator checks all the read operations and determines whether the final `tts` is within the lease. For a read operation whose lease is not valid (i.e., $\texttt{tts} > read.rts$), the coordinator posts an RDMA `READ` to fetch the metadata of the tuple again from the corresponding participant in step **b3**. If there is an update of `rts`, an RDMA `WRITE` is performed.

4.6 CALVIN

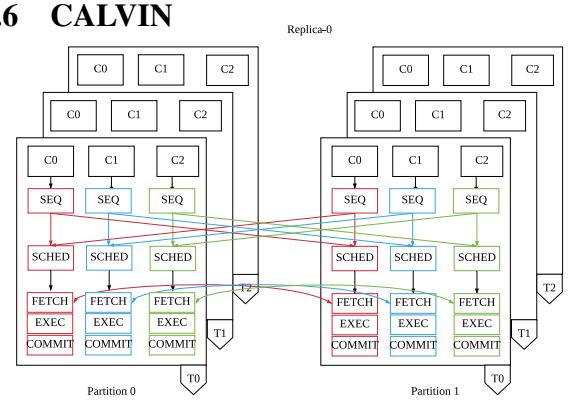

Figure 11: CALVIN Overview

CALVIN [28] allows transactions to execute in a deterministic manner. It was initially designed to reduce transaction aborts due to non-deterministic events such as node failures. By replicating transaction requests across multiple replicas and executing transactions in these replicas in parallel, CALVIN ensures that all replicas maintain a consistent state at any time of transaction execution and thus any replica is capable of serving as an immediate replacement of a failed node.

At a high level, CALVIN includes a *sequencing* layer and a *scheduling* layer, both distributed in different partitions. The sequencing layer intercepts and batches a sequence of transaction inputs, and sends them to the scheduling layer at other partitions for every epoch. After collecting transaction inputs from the sequencing layer of all partitions and forming a deterministic order, the scheduling layer orchestrates transaction executions sequentially according to the order. Figure 11 shows a possible configuration of CALVIN in RCC, which contains one replica, two partitions, three threads, and three co-routines. Compared to CALVIN's original architecture in [28], we made two modifications: 1)

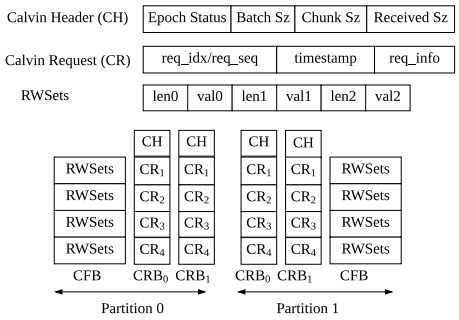

Figure 12: RDMA-enabled buffer organization for one CALVIN co-routine with batch size per epoch = 4 and the maximum number of read/write sets supported per transaction = 3.

To focus on the concurrency control, we assume a reliable cluster and omit replicating transaction inputs across multiple replicas; 2) To allow CALVIN to scale horizontally to multiple co-routines, each co-routine serves as a sequencer (SEQ) and a scheduler (SCHED) of a batch of transactions. After collecting all batches as a scheduler, each co-routine starts fetching, executing, and committing transactions for that epoch.

We define the *counterpart* of a coordinator co-routine as another remote co-routine with the same co-routine ID and thread ID. In CALVIN, two types of inter-partition communications are required: 1) Transaction input broadcasting. Upon batching transaction requests in one epoch, a coordinator broadcasts its batched transaction inputs at the sequencing stage to its counterparts at all partitions at the scheduling stage; and 2) Value forwarding, where a coordinator at the fetching stage forwards its local reads to all counterparts.

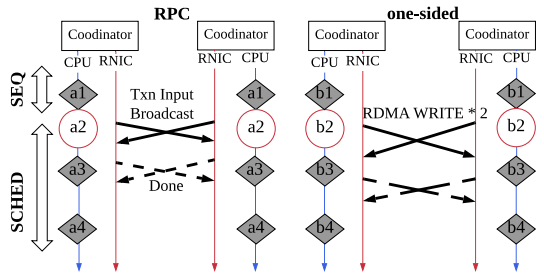

Figure 13: CALVIN: Sequencing and Scheduling

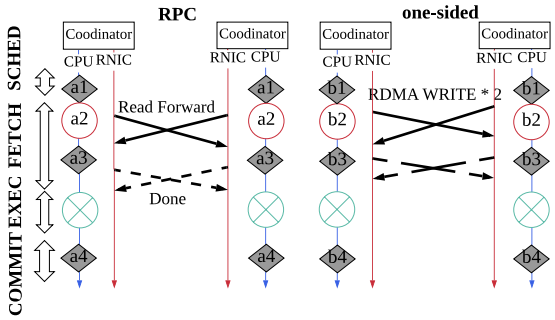

Figure 14: CALVIN: Fetch, Execute and Commit

RCC CALVIN uses two memory buffers that enable RDMA remote access — CALVIN Request Buffer (CRB) and CALVIN Forward Buffer (CFB). Each CRB is an RDMA-enabled memory region containing one CALVIN Header (CH) and a list of CALVIN Requests (CR). Every co-routine contains as

many CRBs (indexed from 0 to $partition-1$) as the number of partitions to collect transaction requests from all counterparts. It also includes one CFB to be used for receiving forwarded values from counterparts. Each CH has control information for CALVIN's deterministic executor to decide whether the scheduler has collected all transaction inputs in one epoch and whether all transactions in a batch have finished execution and the executor should move on to the next epoch. Figure 12 shows an example buffer organization for each co-routine.

Figure 13 shows the procedure of transaction input broadcasting. For **RPC**, at step **a1**, a coordinator at partition i first accumulates transaction requests at CRB_i, and then broadcasts a batch of transaction requests to other partition nodes using an RPC (step **a2**). The handler of each receiving node then buffers the batch at CRB_i of its own memory (step **a3**). After receiving each batch from other counterparts, the coordinator then sorts all the received transaction requests based on their timestamps (**a4**). For **one-sided**, the local operations (step **b1**, step **b4**) are similar. For step **b2**, a coordinator first posts a one-sided RDMA `WRITE` to write all local CALVIN requests starting from the offset of CR_1 of CRB_i. Then it posts the second one-sided RDMA `WRITE` to update the remote *Received Sz* field in the corresponding CALVIN header using the offset of CRB_i. Note that all offsets are pre-calculated and do not incur an extra runtime penalty. step **b3** requires the receiving counterpart to wait until all batches of transactions are collected; it watches the *Received Sz* field of the CALVIN header and yields until it captures the all-batch-received event. All counterparts of a scheduler may be executing different transactions at the same time, but they execute in a lockstep manner at epoch granularity.

Figure 14 shows the execution of one transaction in an epoch. Note that a transaction, although sequenced by only one coordinator, runs at multiple machines after copied to other counterparts in the scheduling stage. Each transaction only requests locks for its local records (step **a1** and step **b1**). Upon failure, the transaction aborts and retries. After successfully locking and reading all local records, a transaction starts forwarding all local records to active counterparts (i.e., counterparts that need the forwarded values) (step **a2** and step **b2**). The CFB is necessary since the forwarded values can not go directly into the actual read/write sets of other counterparts. Without CFB, they would mess up the read/write sets of in-flight transactions in the same epoch in the other counterparts.

For **RPC** implementation, after receiving forward values of transaction i in the epoch, the handler will buffer the values locally at CFB_i, waiting for the counterpart to pick up the values when needed (step **a3**). For **one-sided** implementation, to ensure correctness, two RDMA `WRITE`s are used, one being writing the actual forwarded value to its offset on the counterpart machine, followed by the other writing the length of the value (step **b3**). The counterpart consistently checks the length field to see whether the forwarded values needed are already installed and yield otherwise. Both RPC and one-sided implementations share the same commit stage (step **a4** and step **b4**): each coordinator commits their local writes locally without necessarily reaching out to other machines.

5. EVALUATION

5.1 Workloads

We use three popular OLTP benchmarks, SmallBank [27], YCSB [6], and TPC-C [8], to test the performance of protocols using two-sided RPC (denoted as **RPC**) or one-sided primitives (denoted as **onesided**). Records are partitioned across nodes. To eliminate the effects of locality, all transactions use network operations to fetch and update the data.

SmallBank [27] is a simple banking application. Each transaction performs simple reads and writes on the account data. SmallBank features a small number of writes and reads in one transaction (< 3) with simple arithmetic operations. This makes SmallBank a network-intensive application.

YCSB [6] (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 and the skews are all configurable. In all our experiments, the record length is set to 64 bytes, with 1200000 entries in the table. We used hot area in YCSB, which consists of 1200 entries, i.e., 0.1% of total records, to control contention. The number of entries and the hot area we use is proportional to the number of threads in RCC. There are 10 operations in one transaction, with different read/write ratios.

TPC-C [8] simulates the processing of warehouse orders. In our evaluation, we run the transaction "New-Order", which consists of longer (up to 15) write sets and more complex transaction executions. In this benchmark, TPC-C has higher CPU utilization than SmallBank.

5.2 Execution Setup

We test our framework on two clusters. The first is an RDMA-capable cluster with eight nodes, each equipped with two 12-core Intel Xeon E5-2670 v3 processor, 128GB RAM, and one ConnectX-4 EDR 100Gb/s InfiniBand MT27700. The second cluster is an RDMA-capable cluster with 16 nodes. Each node has two 8-core Intel Xeon CPU E5-2630 v3 processors, 64GB RAM, and one ConnectX-3 Pro FDR 56Gb/s InfiniBand MT27520. As there is only one RNIC on each node, we only run evaluations on the CPU on the same NUMA node with the RNIC to prevent NUMA from affecting our results. We name the first cluster EDR and the second cluster FDR in our experiments. We evaluate both **RPC** and **onesided** implementations of all RCC protocols and focus on three metrics: *throughput, latency, and abort rate*. To concentrate on comparing communication styles and protocols, we enforce coordinators to self-generate transaction requests.

5.3 Overall Results

Figure 15 shows the overall results. In overall evaluations of YCSB, we have 20% write, and 80% read and set the computation in the execution phase of YCSB to 5% of the total latency of a transaction to simulate real-world applications. For evaluating throughput and abort rate, on the EDR cluster, we use ten threads on each node and ten co-routines for each thread, each co-routine producing and handling transaction requests independently. For the FDR cluster, we use eight threads due to the limited number of cores in one CPU.

Finding 1: **Onesided** has an equal or higher throughput than **RPC** with only two exceptions for MVCC on EDR.

We attribute *Finding 1* to the fact that **onesided** can bypass remote CPUs, thus having a smaller communica-

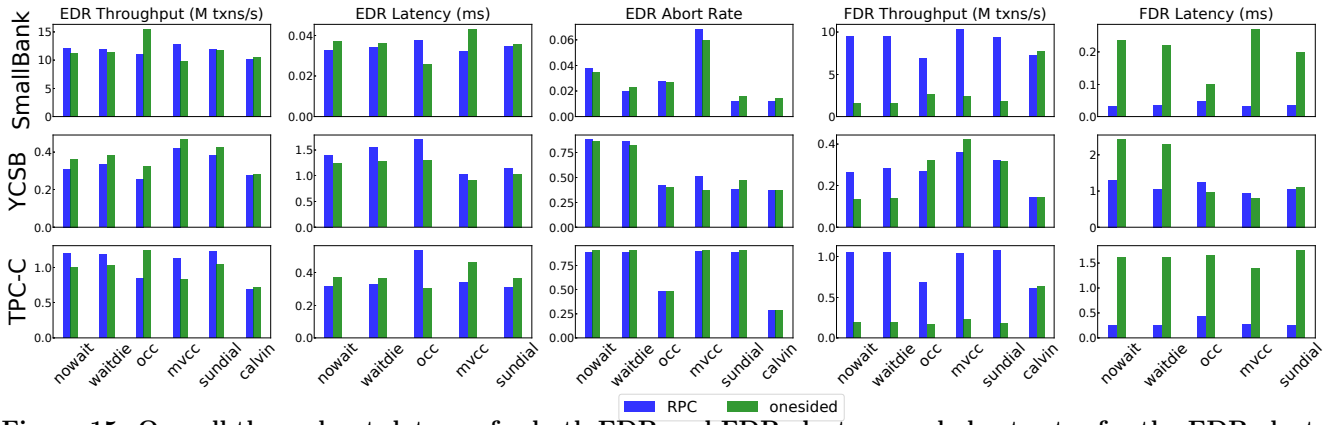

Figure 15: Overall throughput, latency for both EDR and FDR clusters, and abort rates for the EDR cluster

tion overhead in general compared to their **RPC** counterparts for each round trip. This observation is true only when **onesided** implementations do not incur an excessive amount of communication round trips compared to their **RPC** counterparts. However, this is not true for MVCC. From the SmallBank experiment results in Figure 16, **onesided** MVCC incur 72% more communication round trips while other protocols incur only a range of 6% to 34% more. Therefore **RPC** is better than **onesided** for MVCC on SmallBank and TPC-C. As for MVCC on computation-intensive YCSB, **onesided** still dominates **RPC** even with 72% more round trips; the advantage of bypassing remote CPUs becomes salient when remote CPUs are kept busy with computations.

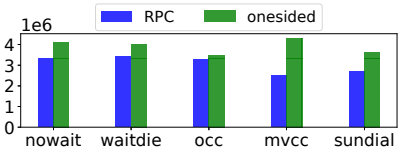

Figure 16: Network Round Trips

Finding 2: While **onesided** is generally better than **RPC** on the EDR, **onesided** OCC is not necessarily the best choice.

Protocols can be compared easily in RCC. For SmallBank, which is a network-intensive benchmark and of low contention, **onesided** OCC is the best choice because it has the least number of network operations due to its optimistic assumption. For YCSB, which has more computation in the execution stage and more operations within one transaction, **onesided** MVCC is the best. Moreover, it resolves many read-write conflicts by maintaining multiple versions of the records. So it has the lowest abort rate and highest throughput. For TPC-C, **onesided** SUNDIAL is the best since it has the dynamic read lease and transaction timestamp.

Finding 3: One-sided operations are not beneficial with low bandwidth (56Gb/s FDR). Only two protocols benefit from one-sided primitives under a computation-intensive workload.

On the FDR cluster, for read, write and lock requests, the latency of one-sided operations is 2.3x more than RPC with two-sided operations. The throughput and latency results of SmallBank shows that **RPC** is much better than **onesided** when the application is network bounded. YCSB, while having additional computation in the execution stage to increase co-routine execution time and burden remote CPUs, can still take advantage of the feature of **onesided** to bypass remote CPU and to outperform **RPC** for some protocols like MVCC and OCC when the application is CPU-bounded. We do all experiments on the EDR cluster for later subsections.

In both clusters, the latency results essentially reflect similar conclusions as we have drawn from throughput results. As we use multiple (i.e., ten) co-routines to improve throughput, the latency of executing one transaction is adversely affected due to context switching. We will investigate the latency breakdown by using just one co-routine later.

5.4 Impact of Co-routines

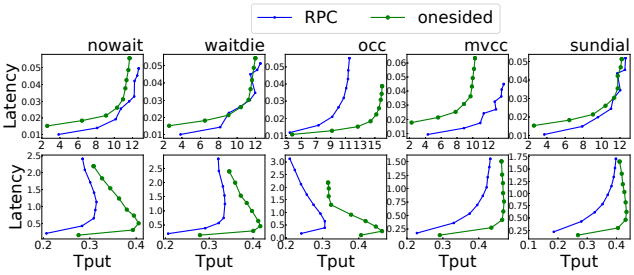

Figure 17: Throughput (M txns/s) and Latency (ms) for SmallBank (Up) and YCSB (Down) with co-routines ranging from 1 to 17 and a step of 2.

To understand how co-routines affect the performance, we increase the co-routines from 1 to 17/19 with a step of 2 to test the latency and throughput in both SmallBank and YCSB. Results are reported on the EDR cluster.

Finding 4: The number of co-routines has a great impact on the throughput and latency of both **RPC** and **onesided** protocols while the best number of co-routines vary for different protocol implementations and different workloads. RCC can provide a clear conclusion in making this design decision when building new transaction systems.

As seen in Figure 17, on one hand, increasing the number of co-routines can help hide the latency of network operations, thus may improve throughput to some point. On

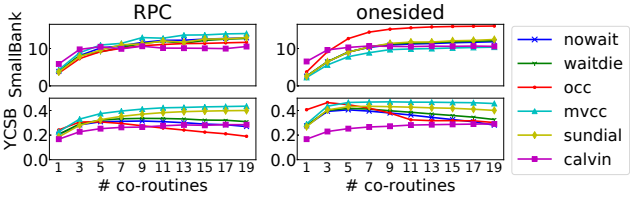

Figure 18: The impact of #co-routine on throughput (M txns/s)

the other hand, it will largely increase the latency of each transaction, leading to higher contention. This is because the tuples are locked for a longer time (2PL) or more likely to be interrupted for its longer life span (OCC). These two factors have different effects on SmallBank and YCSB.

As seen in Figure 18, for SmallBank, the throughput of both **RPC** and **onesided** grow with the number of co-routines. As SmallBank is a network-intensive application, co-routines can greatly improve the throughput by overlapping the waiting time of network operations with transaction execution. As SmallBank touches few tuples in one transaction, there are even fewer tuples locked at the same time for NOWAIT and WAITDIE, and the transaction has a shorter life span for OCC. Overall, the overlapping serves as the principal reason why throughput grows as RCC uses an increasing number of co-routines. **Onesided** OCC has the most rapid growth in throughput because it locks records only at commit time, thus incurring the least conflict as co-routines increase. For **onesided** MVCC and SUNDIAL, which use more network operations for lower abort rate, the throughput is lower than that of the trivial protocols efficiently implemented in **onesided**.

For YCSB, which has more operations (i.e., 10) compared to SmallBank, more tuples are locked at the same time. Meanwhile, each transaction takes more time to finish, making records be locked even longer. Under this scenario, using more co-routines may significantly increase contention and adversely affect throughput after a critical point for some protocols, as seen in Figure 17 and Figure 18. Among **RPC** implementations of all protocols, for those that can better handle read-write conflicts, i.e., MVCC and SUNDIAL, their throughput can continue growing as the number of co-routines increases. Recall that MVCC keeps old versions for read operations and that SUNDIAL dynamically chooses the transaction timestamp and updates the read lease. But protocols like NOWAIT and WAITDIE, which lock all tuples involved in read and write operations, suffer from longer locked tuples with more co-routines. OCC is designed for the low contention scenario. As we mentioned before, with more computation in the execution stage OCC has higher abort cost. So increasing the number of co-routines exacerbates its performance. CALVIN performance is mainly bounded by the computation, because each coordinator that actively participates in a transaction has to perform the time-consuming execution stage even the transaction was not initiated by its own sequencing stage, leading to lower effective throughput.

Comparing the execution of NOWAIT, WAITDIE and OCC implementations on SmallBank and YCSB, we observe that the optimal co-routine number gets smaller as a transaction touches more records. RCC can draw useful conclusions as such for future RDMA-based transaction system builders when they consider optimizing system throughput.

5.5 Impact of Contention Level

To show how well RDMA-based protocols can handle contention, we compare the throughput of six protocols with different contention levels for YCSB. We control the contention levels by varying the possibility of one read or write visiting a small portion of data records, i.e., hot area. The results (on EDR cluster) are shown in Figure 19.

Finding 5: The throughput of different protocol implementations drop at different rates for **RPC** and **onesided**. MVCC *always* outperforms the state-of-the-art SUNDIAL irrespective of contention levels.

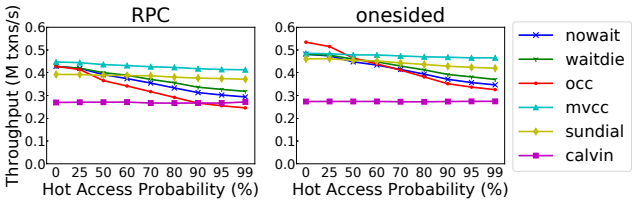

Figure 19: The impact of contention level to throughput for YCSB

In both **RPC** and **onesided** implementations, the performance of OCC decreases most sharply 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 significantly decrease due to the intensive conflict read and write locks. MVCC and SUNDIAL are less affected when the conflict rate increases; their throughput decrease is slow because they have optimization to avoid read-write conflict. CALVIN is not affected much by different contention levels because a CALVIN transaction only locks local records for a short period without necessarily waiting for remote locks. This behavior of CALVIN is confirmed in [28].

We also notice that **onesided** SUNDIAL and MVCC, although featuring good read-write conflict management, are worse than **onesided** OCC when a transaction has the least conflict rate. That is because these two have more complicated operations to maintain more information to reduce the abort rate, which is more costly because every access to remote data will trigger network operation in their **onesided** versions.

It is worthwhile to note that MVCC is consistently better than SUNDIAL irrespective of contention levels in both **RPC** and **onesided** implementations for YCSB. While we do not expect [35] to compare with all previous concurrency control protocols under all possible workloads, our RCC's results show a strong indication that SUNDIAL is worse than one variant of MVCC implementations when the workload is computation-intensive in the RDMA context. Table 3 shows a reference comparison of our version of SUNDIAL with the [35]. This non-trivial observation would not be possible without our unified RDMA-based concurrency control framework.

5.6 Impact of Execution Workloads

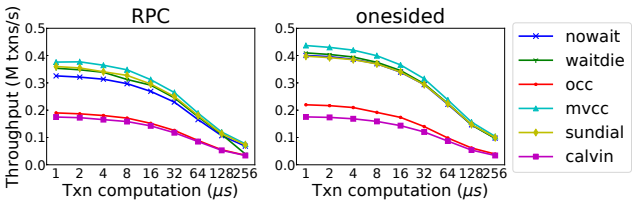

Figure 20: The impact of computation workload on throughput

To study the implication of computation ratio, we add dummy computation in the execution stage of YCSB, ranging from 1 to 256 μs. We carry out experiments on the EDR cluster and show results in Figure 20.

Finding 6: **RPC** and **onesided** share a similar decreasing trend as computation increases at the execution stage; **onesided** always outperforms **RPC** for different computation workloads.

In **RPC** versions, more computation time on the execution stage causes higher latency for an RPC request to get handled, which significantly increases the time that a tuple is locked, leading to a higher abort rate. **onesided** versions are also adversely affected yet have a relatively slower decreasing trend. Suffering from both high abort cost and long computation time, OCC performs much worse than 2PL protocols. CALVIN's performance is bounded by local computation; thus, there is no difference between its **RPC** and **onesided** version.

5.7 System Resource Utilization

Finding 7: **Onesided** implementations are less likely to be bottlenecked by system resources and can reach much higher network utilization than their **RPC** counterparts.

We measure system resource utilization of both **RPC** and **onesided** implementations by two experiments. First, we use two different thread numbers (i.e., five threads and ten threads) of all implementations on the EDR cluster for the YCSB benchmark and show the throughput increase in Table 1. Since our RCC is symmetric: i.e., each co-routine in some thread only communicates with another co-routine at the *same thread* of another machine. Ideally, doubling thread numbers would also double the throughput if not bottlenecked by system resources and if we did not consider the effect of an increased contention level. In reality, from Table 1, we can see that **onesided** implementations are equal or closer to the ideal case, thus having equal or less likelihood of being bottlenecked by system resources as **RPC** implementations. This behavior is understandable since **onesided** implementations bypass system calls and kernels, thus avoiding many possibilities of being bottlenecked. **Onesided** SUNDIAL is the protocol that is closest to linear scalability in RCC. Overall, RCC enables researchers to compare RDMA-based protocols in terms of system utilization. We leave pinpointing the exact system resource bottleneck as an important future work.

Second, to characterize the network utilization of different protocols in both **RPC** and **onesided** implementations, we measure their message rates in RCC when executing the YCSB workload on the EDR cluster. Table 2 shows our results. We observe the max message rate in RCC reaches 31.06 Mpps by **onesided** NOWAIT. Given that the RDMA peak message rate of our EDR cluster is 33 Mpps, **onesided** NOWAIT has reached 94.1% of the peak. On average **onesided** implementations send approximately *3x* more packages per second than their **RPC** counterparts, which indicates that **onesided** protocols all have much higher network utilization.

5.8 Stage Latency Breakdown

While we have already done some experiments to compare various aspects of different protocols in RCC. Previous latency results are still not an accurate reflection of the actual bottleneck of each protocol since interleaving transactions among multiple co-routines increases transaction latency nondeterministically. To pinpoint the stage-wise bottleneck of each protocol implementation further, we run all implementations except CALVIN using *only one* co-routine for SmallBank on 4 nodes of EDR cluster and break down a whole transaction latency into different stages: *Read*, *Lock*, *Release*, *Commit*. SUNDIAL has an extra lease *Renew* latency. CALVIN is not included in this experiment because all these stages are local operations for CALVIN. Figure 21 shows the stage-wise latency breakdown. Note that a record is returned together with locking result (as a whole tuple) for **RPC** NOWAIT and WAITDIE, so their *Read* latency is combined into the *Lock* latency.

Table 1: Throughput in K/s for different threads on EDR

	RPC			**onesided**		
#threads	5	10		5	10	
nowait	40.23	67.9	1.69x	56.83	107.4	1.89x
waitdie	40.12	71.69	1.79x	57.26	102.1	1.78x
occ	27.45	48.09	1.75x	47.93	84.53	1.76x
mvcc	52.34	91.93	1.76x	68.16	125.6	1.84x
sundial	50.28	80.68	1.60x	60.26	119.2	1.98x

Table 2: Msg. Rate in M pkgs/s on EDR and utilization

	RPC	**onesided**	network utilization %
nowait	9.49	31.06	28.8 → 94.1
waitdie	9.25	30.77	28.0 → 93.2
occ	9.75	30.38	29.5 → 92.1
mvcc	10.70	28.46	32.4 → 86.2
sundial	9.91	29.52	30.0 → 89.5

For **RPC** implementations, as illustrated in Figure 21, MVCC incurs the largest *Read* latency due to the participant's traversal of all `wts` of the requested record. MVCC also incurs the largest *Lock* latency due to the participant's double-checking of conditions for successful locking. OCC incurs the largest latency at *Commit* and *Release* stages and second-largest *Read* latency compared to other protocols. This is because the OCC implementation of [31] accumulates all RPC read/write messages in one stage into one message before broadcasting it to all participants, which uses higher network bandwidth and incurs larger latency (This latency is not apparent with ten co-routines as in Figure 15).

For **onesided** implementations as illustrated in Figure 21, *Release* is not the bottleneck for any protocol. MVCC and SUNDIAL suffer from the top-2 *Read* latency due to their extra logic for ensuring the validity of record versions or lease upon read operations. MVCC suffers from the largest *Lock* latency since it needs to retrieve longer tuple upon locking.

By analyzing stage latency results, researchers can make use of RCC to understand better and mitigate the stage-wise bottlenecks of **RPC** and **onesided** implementations.

5.9 Comparison with the State-of-the-art

As we have stated that RCC is the *first* unified framework of RDMA-based implementations of representative protocols. There is no apple-to-apple comparison for each protocol. We implemented five TCP-based RPC versions in RCC and compared these TCP protocols in RCC with corresponding TCP-based protocols in [14] and [35] using similar configurations on YCSB. Table 3 shows the results. As we can see, except SUNDIAL, our TCP implementations are comparable or better than corresponding implementations in [14]. The two-sided RDMA versions in RCC perform an order of magnitude better than corresponding TCP versions in RCC or [14].

5.10 Scalability and implications to architecture design

Table 3: Comparing throughput (Txns/s) of RCC with the State-of-the-Art protocol implementations using TCP. a) RCC TCP-based. b) RCC two-sided.

	Other TCP-based	a)	b)
nowait	approx. 50000 [14]	48394	308177
waitdie	approx. 10000 [14]	51444	336514
occ	approx. 40000 [14]	49949	257307
mvcc	approx. 20000 [14]	63514	420000
sundial	approx. 70000 [35]	38275	383237

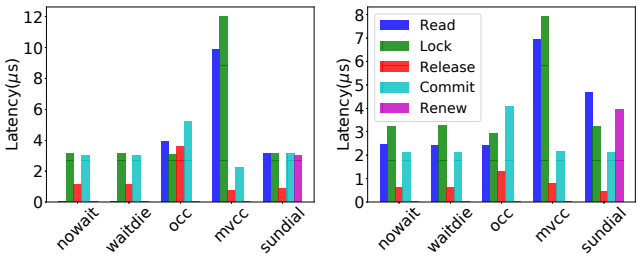

Figure 21: Latency breakdown for RPC (Left) and onesided (Right)

To understand how protocols perform with a much larger cluster, we run all protocols against the YCSB benchmark (10% write and $5\mu s$ computation time) on several emulated larger EDR clusters, each equipped with enough QPs for $4 * (10x + 9)$ number of fully connected nodes where $x = 0, 1, 2, .., 9$. shown in Figure 22. Note that UD QPs are per-core for two-sided RPC, while RC QPs are per-core and per-connection for one-sided primitives. Each RDMA op uses multiple QPs in a round-robin manner, similar to [31].

We can see that on the emulated larger EDR clusters, one-sided implementations maintain their superiority over RPC implementations on large clusters and up to a 396-node cluster, yet the advantages are gradually reduced as cluster size increases. MVCC still dominates SUNDIAL by 9% to 12%.

Figure 22 envisions a trend for large RDMA systems that an increasing number of QPs needed for larger clusters will cause performance loss due to limited NIC capabilities. As [17] has pointed out that NIC cache miss is an essential factor contributing to the performance loss of large-scale RDMA transaction systems. We attribute the performance degradation of all protocols for one-sided implementations to QP state cache miss in the NIC SRAM. It can also be seen that the initial performance drop from 4 nodes to 36 nodes (i.e., $x = 1$) is much higher than that of even larger clusters (i.e., $x > 1$), which implies that NIC QPs are purged into DRAM frequently with 36 nodes or more. Therefore, for large clusters greater than 36 nodes, the architecture of RNIC may need to leverage multi-level caches to reduce the number of QP state cache misses. Meanwhile, we can see that the absolute performance drop for each protocol is roughly the same as the number of nodes in a cluster grows. However, as MVCC performs the best, MVCC is the *least* sensitive protocol as more QPs are evicted from the NIC cache. The benefit of knowing this information comes in two folds. First, protocol developers can understand which protocol is the most friendly (i.e., Non-susceptible to performance drop due to NIC QP miss) when given an RDMA-capable cluster of a certain scale. Second, architecture designers are

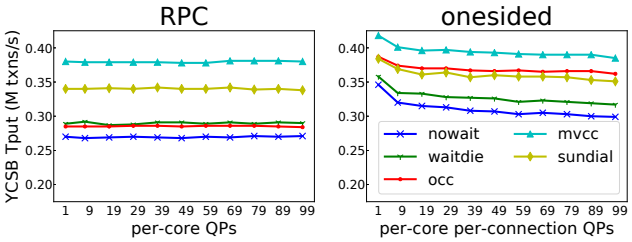

Figure 22: Throughput on emulated large EDR clusters.

better able to identify the bottlenecks revealed by sensitive one-sided protocol implementations.

6. RELATED WORK

Comparisons among concurrency control protocols [1] uses modeling techniques to reveal the hidden connections between protocols' underlying assumptions as well as their seemingly contradictory performance results. [15] compares three concurrency control protocols in real-time database systems but only restraints to optimistic ones. [33, 34] focuses on the scalability issues and examines seven concurrency control protocols on a main-memory DBMS on top of a simulated 1024-core system. Deneva [14] is the recent work comparing distributed concurrency control protocols in a single unified framework. RCC takes the first step in comparing different protocols under the context of various RDMA primitives.

Comparisons between RDMA primitives [19] compares the use of RDMA `WRITE` and RDMA `READ` when constructing a high performance key-value system. [11] finds out that RDMA `WRITE`'s polling significantly outperforms `SEND` and `RECV` verbs when constructing the FaRM's communication subsystem. [18] shows that UD-based RPC using `SEND` and `RECV` outperforms one-sided primitives. [31] did more primitive-level comparisons with different payload sizes. Compared to them, RCC compares the primitives with a much wider range of concurrency control algorithms on two clusters with different RDMA capabilities.

Distributed transaction systems High performance transaction systems have been investigated intensively [28, 7, 30, 12, 5, 31, 21]. Most of them focus on distributed transaction systems [7, 12, 5, 31] since it is more challenging to implement a high performance transaction system with data partitioned across the nodes. Some works, e.g., [21, 12, 5, 31, 18], focus only on one protocol (i.e., some variants of OCC). Other works like [32, 35, 28] explore novel techniques like determinism or leasing. However, these works did not explore the opportunity of using RDMA networks.

7. CONCLUSION

In this paper, we build *RCC*, the first unified RDMA-enabled distributed transaction processing framework supporting six concurrency control protocols using either two-sided or one-sided primitives. We intensively optimize the performance using techniques such as co-routines, outstanding requests, and doorbell batching. Based on RCC, we conduct the first and most comprehensive (to the best of our knowledge) comparison of different representative distributed concurrency control protocols on two clusters with different RDMA network capabilities.

8. REFERENCES

[1] R. Agrawal, M. J. Carey, and M. Livny. Concurrency control performance modeling: Alternatives and implications. *ACM Transactions on Database Systems (TODS)*, 12(4):609–654, 1987.

[2] 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*, 8(3):185–196, 2014.

[3] P. A. Bernstein, P. A. Bernstein, and N. Goodman. Concurrency control in distributed database systems. *ACM Comput. Surv.*, 13(2):185–221, June 1981.

[4] P. A. Bernstein and N. Goodman. Multiversion concurrency control—theory and algorithms. *ACM Transactions on Database Systems (TODS)*, 8(4):465–483, 1983.

[5] 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*, page 26. ACM, 2016.

[6] 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*, pages 143–154. ACM, 2010.

[7] 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)*, 31(3):8, 2013.

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

[9] 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)*, pages 223–235, 2012.

[10] C. Curino, E. Jones, Y. Zhang, and S. Madden. Schism: a workload-driven approach to database replication and partitioning. *Proceedings of the VLDB Endowment*, 3(1-2):48–57, 2010.

[11] 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)*, pages 401–414, 2014.

[12] 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*, pages 54–70. ACM, 2015.

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

[14] R. Harding, D. Van Aken, A. Pavlo, and M. Stonebraker. An evaluation of distributed concurrency control. *Proceedings of the VLDB Endowment*, 10(5):553–564, 2017.

[15] J. Huang, J. A. Stankovic, K. Ramamritham, and D. F. Towsley. Experimental evaluation of real-time optimistic concurrency control schemes. In *VLDB*, volume 91, pages 35–46, 1991.

[16] A. Kalia, M. Kaminsky, and D. G. Andersen. Using rdma efficiently for key-value services. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, page 295–306, New York, NY, USA, 2014. Association for Computing Machinery.

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

[18] 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)*, pages 185–201, 2016.

[19] A. K. M. Kaminsky and D. G. Andersen. Using RDMA efficiently for key-value services. In *Proc. of SIGCOMM*, 2014.

[20] H.-T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *ACM Transactions on Database Systems (TODS)*, 6(2):213–226, 1981.

[21] 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*, pages 71–86. ACM, 2015.

[22] 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*, 7(5):329–340, 2014.

[23] 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)*, pages 479–494, 2014.

[24] 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*, pages 61–72. ACM, 2012.

[25] 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*, pages 1311–1326. ACM, 2015.

[26] M. Stonebraker. The case for shared nothing. *IEEE Database Eng. Bull.*, 9(1):4–9, 1986.

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

[28] 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*, SIGMOD '12, pages 1–12, New York, NY, USA, 2012. ACM.

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

[30] 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*, pages 18–32. ACM, 2013.

[31] 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)*, pages 233–251, 2018.

[32] 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*, pages 87–104. ACM, 2015.

[33] X. Yu. *An evaluation of concurrency control with one thousand cores*. PhD thesis, Massachusetts Institute of Technology, 2015.

[34] 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*, 8(3):209–220, 2014.

[35] 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*, 11(10):1289–1302, 2018.