Extracting More Concurrency from Distributed Transactions

Shuai Mu, *Tsinghua University and New York University;* Yang Cui and Yang Zhang,
New York University; Wyatt Lloyd, *University of Southern California and Facebook, Inc.;*
Jinyang Li, *New York University*

**This paper is included in the Proceedings of the
11th USENIX Symposium on
Operating Systems Design and Implementation.**

October 6–8, 2014 • Broomfield, CO

Extracting More Concurrency from Distributed Transactions

Shuai Mu[†‡], Yang Cui[‡], Yang Zhang[‡], Wyatt Lloyd[♯♮], Jinyang Li[‡]

[†]*Tsinghua University,* * [‡]*New York University,* [♯]*University of Southern California,* [♮]*Facebook*

Abstract

Distributed storage systems run transactions across machines to ensure serializability. Traditional protocols for distributed transactions are based on two-phase locking (2PL) or optimistic concurrency control (OCC). 2PL serializes transactions as soon as they conflict and OCC resorts to aborts, leaving many opportunities for concurrency on the table. This paper presents ROCOCO, a novel concurrency control protocol for distributed transactions that outperforms 2PL and OCC by allowing more concurrency. ROCOCO executes a transaction as a collection of atomic pieces, each of which commonly involves only a single server. Servers first track dependencies between concurrent transactions without actually executing them. At commit time, a transaction's dependency information is sent to all servers so they can re-order conflicting pieces and execute them in a serializable order.

We compare ROCOCO to OCC and 2PL using a scaled TPC-C benchmark. ROCOCO outperforms 2PL and OCC in workloads with varying degrees of contention. When the contention is high, ROCOCO's throughput is 130% and 347% higher than that of 2PL and OCC.

1 Introduction

Many large-scale Web services, such as Amazon, rely on a distributed online transaction processing (OLTP) system as their storage backend. OLTP systems require concurrency control to guarantee strict serializability [12, 13], so that websites running on top of them can function correctly. Without strong concurrency control, sites could sell items that are out of stock, deliver items to customers twice, double-charge a customer for a sale, or indicate to a customer they did not purchase an item they actually did.

While concurrency control is a well-studied field, traditional protocols such as two-phase locking (2PL) [12] and optimistic concurrency control (OCC) [36] perform poorly when workloads exhibit a non-trivial amount of

---

contention [8, 30]. The performance drop is particularly pronounced when running these protocols in a distributed setting. When there are many conflicting concurrent transactions, 2PL and OCC abort and retry many of them, leading to low throughput and high latency. In our evaluation in § 5, the throughput of 2PL and OCC drops to less than 10% of its maximum as contention increases.

Unfortunately, contention is not rare in large-scale OLTP applications. For example, consider a transaction where a customer purchases a few items from a shopping website. Concurrent purchases by different customers on the same item create conflicts. Moreover, as the system scales—i.e., the site becomes more popular and has more customers, but maintains a relatively stable set of items—concurrent purchases to the same item are more likely to happen, leading to a greater contention rate.

In this paper we presents ROCOCO (ReOrdering COnflicts for COncurrency), a distributed concurrency control protocol that extracts more concurrency under contended workload than previous approaches. ROCOCO achieves safe interleavings without aborting or blocking transactions using two key techniques: 1) deferred and reordered execution using dependency tracking [38, 46]; and 2) offline safety checking based on the theory of transaction chopping [50, 49, 57].

ROCOCO is a two round protocol that executes transactions that have been structured into a collection of atomic pieces, each typically involving data access on a single server. A set of coordinators run the protocol on behalf of clients. The first phase distributes the pieces to the appropriate servers and establishes a provisional order of execution on each server. Servers typically defer execution of the pieces until the second round so they can be reordered if necessary. Servers complete the first phase by replying to the coordinator with dependency information that indicates the order of arrival for conflicting pieces of different transactions.

The coordinator aggregates this dependency information and distributes it to all involved servers. Servers use the aggregated dependency information to recognize if the pieces of concurrent transactions arrived at servers in a strictly serializable order in the first phase. If so, they execute pieces in that order in the second phase. If not, servers reorder the pieces deterministically and then exe-

---

cute them. In both cases, ROCOCO is able to avoid aborts and commits all transactions.

Dependencies are usually exchanged only between servers and coordinators in the two round protocol. But, when conflicting transactions have overlapping but non-identical sets of servers, ROCOCO occasionally requires additional server-to-server communication to ensure a deterministic order.

Not all transaction pieces can have their execution deferred to the second round, e.g., a piece that reads a value to determine what data item to access next. Such pieces must be executed immediately in ROCOCO's first phase, which can result in un-serializable interleavings. To ensure that a strictly serializable reordering is always possible during runtime, ROCOCO performs an offline check on the transaction workload prior to starting the transactions. The offline checker identifies and categorizes potential conflicts. If some pieces of a transaction are found to have unsafe interleaving that cannot be reordered, ROCOCO merges those pieces into a single atomic piece. While a traditional concurrency control protocol is used to execute a merged piece across servers atomically, the ROCOCO protocol is used to execute multiple merged pieces within a transaction.

We implemented ROCOCO and evaluated its performance using a scaled TPC-C benchmark [5]. ROCOCO supports the TPC-C workload without requiring any merged pieces and avoids ever aborting. ROCOCO outperforms 2PL and OCC in workloads with varying degrees of contention. When the contention is high, ROCOCO's throughput is 130% and 347% higher than that of 2PL and OCC. As the system scales across TPC-C warehouse districts and contention increases, the throughput of ROCOCO continues to grow while the throughput of OCC drops to almost zero and 2PL does not scale.

2 Overview

ROCOCO targets OLTP workloads in large-scale distributed database systems, e.g., the backend of e-commerce sites like Amazon. For scalability, database tables are sharded row-wise across multiple servers, with each server holding a subset of certain tables. Thus, a transaction accessing different table rows typically needs to contact more than one server and requires a distributed concurrency control protocol.

For performance, we assume a setup where transactions are executed as *stored procedures*, as in earlier work [34, 52, 26, 28, 41, 56, 55]. Specifically, a distributed transaction consists of a set of stored procedures called *pieces*. Each piece accesses one or more data items stored on a single server using user-defined logic. Thus, each piece can be executed atomically with

```
transaction new_order_fragment:
  #simplified new-order "buys" 1 of item_a,item_b
  input: item_a, item_b
  begin
      ...
  p1:  # reduce stock level of item_a
      R(tab="Stock", key=item_a) → stock
      if (stock > 1):
        W(tab="Stock", key=item_a) ← stock - 1
      ...
  p2:  # reduce stock level of item_b
      R(tab="Stock", key=item_b) → stock
      if (stock > 1):
        W(tab="Stock", key=item_b) ← stock - 1
      ...
  end
```

Figure 1: A fragment of TPC-C new-order transaction containing two pieces.

respect to other concurrent pieces by employing proper local concurrency control. We assume stored procedures are distributed to all servers apriori because they have a minimal storage costs.

2.1 Traditional Approaches Abort Conflicts

Application programmers prefer the strongest isolation level, strict serializability [12, 31], to simplify the reasoning of correctness in the face of concurrent transactions. To guarantee strict serializability, a distributed storage system typically runs standard concurrency control schemes such as two-phase locking (2PL) or optimistic concurrency control (OCC), combined with two-phase commit (2PC) [19].

2PL and OCC perform poorly for contended workload with many conflicting transactions. As an example, consider a simplified fragment of the TPC-C new-order transaction which simulates a customer purchasing two items from a store (Figure 1). The transaction contains two stored procedure pieces, p_1 and p_2, each of which reduces the stock level of a different item. Although each piece can be executed atomically on its server, distributed concurrency control is required to prevent non-serializable interleaving of pieces across servers. For instance, suppose a merchant keeps the same stock level for item a (e.g., a xbox) and item b (e.g., a xbox controller) and always sells the two items as bundles. Without distributed concurrency control, one customer could receive an item a, but not item b, while another customer could receive item b but not item a.

We first examine the behavior of OCC with two transactions, T_1 and T_2. Both purchase the same two items, a and b, that are stored on different servers. Any interleaving of T_1 and T_2's pieces during execution causes aborts when performing OCC validation during 2PC. For example, if T_2 reads the stock level of a after T_1 reads it, but
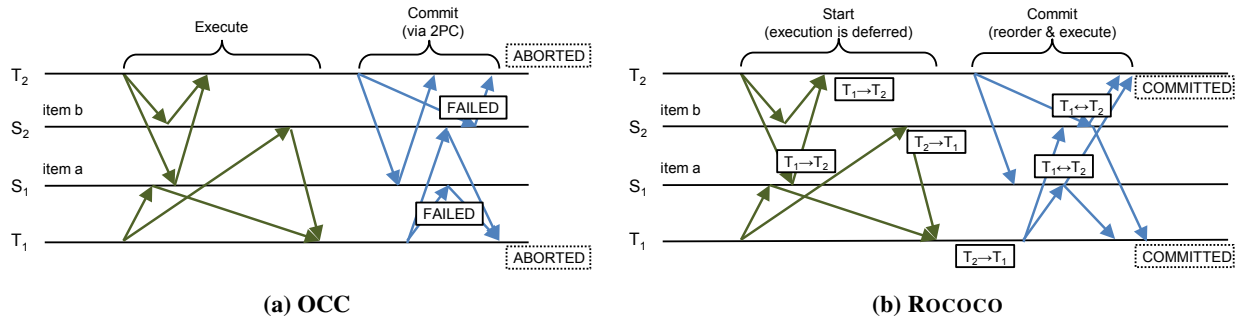
(a) OCC   **(b) ROCOCO**

Figure 2: A possible interleaving of two concurrent new-order transaction fragments. In the left figure (OCC), both transactions fail to validate and abort. In the right figure (ROCOCO), the interference is captured by dependencies and the transactions are reordered to a strictly serializable order before execution.

before T_1 commits its update to a, then T_2 will later fail to validate and abort. Figure 2a shows another example where both T_1 and T_2 are aborted during 2PC because their corresponding 2PC prepare messages are handled by servers in different orders.

2PL outperforms OCC under contention but is still far from satisfactory. 2PL acquires locks for each data access, which serializes the execution of transactions as soon as they perform a conflicting operation. In the new-order example, as soon as T_1 modifies the stock level of a, T_2 is blocked until T_1 completes all of its pieces and commits. In addition to blocking, 2PL also resorts to aborts to prevent deadlocks [19]. As the amount of contention increases, so does the probability of having a deadlock. Furthermore, efficient deadlock prevention mechanisms such as wound-wait [48] have many false positives, thereby causing a large number of aborts even when there is no real deadlock.

2.2 ROCOCO Reorders Conflicts to Commit

Given conditions discussed later, our new concurrency control protocol, ROCOCO, avoids aborting or blocking under contention by identifying and then avoiding interference between transactions. Two transactions *interfere* when executing their constituent pieces in their arrival order at each server would result in a non-serializable execution. For example, T_1 and T_2 interfere in Figure 2b because their pieces arrive in different orders on servers S_1 and S_2. If both pieces of T_1 arrived before both pieces of T_2 they would not interfere.

ROCOCO tracks potential interference using dependency information between pieces of transactions that are generated when pieces *conflict* on a server, i.e., both access the same data location and at least one of them writes to it. Servers use dependency information to detect if transactions interfere and deterministically reorder their pieces so they are executed in the same order on all involved servers and, thus, no longer interfere.

ROCOCO is able to change the order of execution of pieces because it uses two rounds of messages to commit them. The first round starts with a transaction coordinator running on behalf of a client disseminating the pieces of a transaction to the appropriate servers. The servers do not yet execute the pieces and instead return dependency information to the coordinator to complete the first round. The coordinator then combines all the dependency information and distributes it to all the servers in the second round. The servers then reorder pieces of the transaction, if necessary, before executing them.

Figure 2b shows an example of ROCOCO in action. S_1 observes $T_1{\rightarrow}T_2$, reflecting the arrival order of the conflicting pieces it has received from T_1 and T_2. Similarly, S_2 observes $T_2{\rightarrow}T_1$. The coordinator collects $T_1{\leftrightarrow}T_2$ and sends this dependency information to both servers. The servers recognize the cycle of interference and deterministically order the involved transactions and thus their constituent pieces before executing them. The ordering of the two transactions can be any deterministic order, e.g., the order of their globally unique transactions ids, which in the example would execute T_1 and then T_2. With ROCOCO, T_1 and T_2 both commit and neither has to abort or wait for the other.

By reordering interfering transactions instead of aborting them, ROCOCO can achieve significant performance improvement when there is a non-trivial amount of contention, which is often the case with OLTP workloads. For example, a complete TPC-C new order transaction updates a highly contended *order-id* data field as well as 10 purchased items on average. As the number of concurrent requests rises, the probability of contending on a purchased item also increases. Moreover, the power-law distribution often seen in real-world workloads results in even higher contention on "hot" items.

3 Design

The design of ROCOCO includes an offline checker and a runtime protocol. The offline checker determines if the pieces of a collection of transactions can be reordered correctly at runtime. The runtime protocol tracks the dependencies between pieces and reorders their execution if necessary for correctness.

In this section, we explain ROCOCO's offline check (§ 3.1), runtime protocol (§ 3.2), and sketch its correctness (§ 3.3). We then discuss an important optimization (§ 3.4) and the fault tolerance mechanism (§ 3.5).

3.1 Checking When Reordering is Viable

Reordering the execution of pieces of interfering transaction is only possible under certain, common, conditions. This subsection explains the difference between immediate pieces of a transaction that cannot be reordered and deferrable pieces that can. Then it explains how ROCOCO's offline checker uses transaction profiles including immediate/deferrable information to check for the necessary conditions.

Immediate and deferrable pieces. A piece of a transaction is either immediate or deferrable depending on the stored procedure it executes. If the output of a piece p can serve as the input to another piece p', then p is an *immediate* piece because it must be executed before its parent transaction can move on to its subsequent pieces. Conversely, a piece is *deferrable* if its output is not required by any other piece. A server can postpone the execution of a deferrable piece until the commit time of a transaction.

Once executed, immediate pieces cannot be reordered, which can result in a non-serializable interleaving. As an example, suppose p_1 and p_2 in Figure 1 are both immediate instead of deferrable pieces. Then Figure 2's message interleaving makes a total ordering of the transactions impossible. In particular, S_1 executes piece p_1 of T_1 before that of T_2, fixing $T_1 \rightarrow T_2$ in the total order. However, the execution on S_2 fixes $T_2 \rightarrow T_1$ in the total order, a contradiction.

If at least one of the pieces is deferrable, however, a total order can be achieved. For example, instead suppose p_1 is immediate and p_2 is deferrable, then the interleaving can be reordered at S_2 so that p_2 of T_1 is executed before that of T_2, i.e., $T_1 \rightarrow T_2$, which is consistent with the execution at S_1 and thus a total order. ROCOCO's offline checker ensures that there exists such a deferrable piece for all sets of possibly interfering transactions.

The offline checker. In order to ensure that a serializable reordering of conflicting pieces is always possible at runtime, ROCOCO relies on an offline checker that analyzes the conflict profile of *all* transactions to be executed. Fortunately, OLTP workloads typically have a fixed set of transactions that are known apriori [52], making such an offline checker practical.

To build ROCOCO's offline checker, we extend the theory of transaction chopping [50, 57]. For each piece p, we assume the checker knows whether p is an immediate or deferrable piece and the database tables and columns p reads or writes. We do not assume the checker knows which rows p accesses. In our current implementation, programmers explicitly write each transaction as a set of pieces and manually annotate each piece's type and database accesses.

The checker works in several steps. First, it constructs a SC-graph, similar to earlier uses of transaction chopping [50, 57]. Each transaction appears as two instances in the graph where each piece is a vertex and pieces from the same transaction instance are connected by *S(ibling)-edges*. If two pieces access the same database table and at least one of the accesses is a write, they are connected by a *C(onflict)-edge*. If a cycle in the graph contains both S- and C-edges, it is a *SC-cycle*. Each SC-cycle signals a potential conflict that can lead to non-serializable execution [50, 57].

Next, the checker tags each vertex as either an *I(mmediate)* or *D(eferrable)* piece. The checker virally propagates immediacy across C-edge by changing the tag of any piece with a C-edge to an I piece to also be I until there are no C-edges between pieces with different I/D tags. We refer to a C-edge as I-I (or D-D) if both end points are I (or D) pieces. There are no I-D edges.

Finally, the checker examines if there exists an *unreorderable* SC-cycle where all C-edges are I-I edges. If there are none, ROCOCO's basic protocol can safely reorder all conflicts to ensure serializability at runtime. Intuitively, SC-cycles represent potential non-serializable interleavings [49]. However, if an SC-cycle contains at least one D-D edge, ROCOCO can reorder the execution of the D-D edge's pieces to break the cycle, thereby ensuring serializability. For an unreorderable SC-cycle with all I-I C-edges, the checker proposes to merge those pieces in the cycle belonging to the same transaction into a larger atomic piece. In the later section § 4.2, we explain how ROCOCO relies on traditional distributed concurrency control methods such as 2PL or OCC to execute merged pieces.

Figure 3 shows a more complete version of the TPC-C new-order transaction that includes two new pieces in addition to the stock-level-reduction piece discussed earlier. p_1 reads the next order id (`next_oid`), increments it, and writes it back. p_2 modifies the stock level of the purchased item. There may be many instances of p_2, depending on how many items the customer buys, denoted p_2', p_2'', etc. p_3 records the order information in the

```
transaction simplified_new_order:
  input: [item_a, item_b, ...], district d
  begin
        ...
p_1: #pick the next order id
  R(tab="District",col="next_oid",key=d)→ oid
  W(tab="District",col="next_oid",key=d)← oid+1

p_2: #reduce the stock level of each item
   #(one piece for each item)
  R(tab="Stock", key=item)→ stock
  if (stock > 1):
    W(tab="Stock", key=item)← stock-1
  ...

p_3: #add orderline info for each item
   #(one piece for each item)
  W(tab="OrderLine", key=item+oid) ← ...
  ...
  end
```

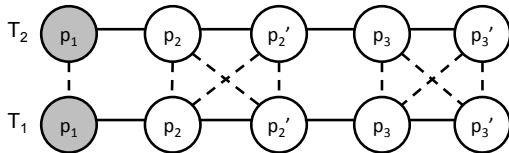**Figure 3: A simplified TPC-C new-order transaction**

Figure 4: SC-graph of the TPC-C new-order sample transaction. Gray circles represent immediate pieces; white circles represent deferrable pieces. Solid lines represent S-edges; dotted lines represent C-edges. ROCOCO can safely execute this transaction workload because all SC-cycles include at least one D-D edge.

database using the order id output by p_1. There may also be multiple instances of p_3, denoted p_3', p_3'', etc. p_1 is an immediate piece because p_3 reads from it while p_2 and p_3 are deferrable pieces. Figure 4 shows the SC-graph of a workload that only contains concurrent new-order transactions that buy two items. ROCOCO can safely execute this workload because all SC-cycles in the graph have a D-D edge.

User-initiated aborts. Previous systems based on transaction chopping [50, 57] sequentially execute pieces and allow user-initiated aborts only in the first piece. RO-COCO, in contrast, executes pieces in parallel so there is no natural "first" piece. For simplicity, we disallow all user-initiated aborts. [1]

3.2 Basic Protocol

ROCOCO's runtime protocol executes a collection of

---

[1] User-initiated aborts are important if the transaction needs to terminate after it has written to the database, which means all writes need rollback. If the aborts happen before any writes, it can be replaced with simple termination.

transactions deemed safe for reordering by the offline checker. Clients delegates the responsibility of coordinating their transactions to separate *coordinator* processes. There can be many coordinators and a typical deployment co-locates coordinators with servers.

Once a coordinator receives a client's transaction request, it processes the transaction in two phases: start and commit. In the *start phase*, the coordinator sends pieces to servers and collects the returned dependency information. In the *commit phase*, the coordinate disseminates the aggregated dependency information to all participating servers who reach a deterministic serializable order to execute conflicting transactions.

Figure 5 shows a typical message flow for ROCOCO.

The start phase. The start phase of a transaction distributes its pieces, sets a provisional order for them on servers, executes immediate pieces, and collects dependency information.

The start phase begins when the coordinator sends out requests for all pieces of a transaction together with their inputs to the appropriate servers—i.e., the servers that store the items read or written by those pieces. If a piece p is immediate, the server will execute p immediately and return its output so that the coordinator can proceed to issue other pieces whose inputs are based on p's output. If p is deferrable, the server buffers it for later execution. The coordinator also parallelizes the issuing of requests when possible, only blocking a request if its inputs are not yet available.

In addition to executing immediate pieces and buffering deferrable ones, each server maintains a dependency graph, dep. Each vertex in dep represents a transaction and its known status, which can be any one in the ordered set {STARTED, COMMITTING, DECIDED}. In addition, for each transaction T involving server S, S keeps a boolean flag $T.finished$ to indicate whether server S has finished committing T. Each edge represents the order of conflicting pieces between two transactions as observed by the server. For example, if a server receives p_1 that writes to data item x. Then, upon receiving p_2 that also accesses x, the server adds a direct edge $p_1.owner{\rightarrow}p_2.owner$ to dep, where $p.owner$ denotes p's corresponding transaction. Moreover, each edge is labeled depending on the types of p_1 and p_2 as immediate or deferred. If both pieces are immediate, the edge is labeled as $\xrightarrow{i}$; if both are deferrable, the edge is $\xrightarrow{d}$. There cannot be an edge between an immediate and deferrable piece because the offline checker eliminated such scenarios when it virally propagated immediacy over C-edges.

Figure 10 summarizes how a server processes a start request in pseudocode. The server returns its updated dep graph and the piece's execution output if the piece is immediate to the coordinator. The coordinator sim-
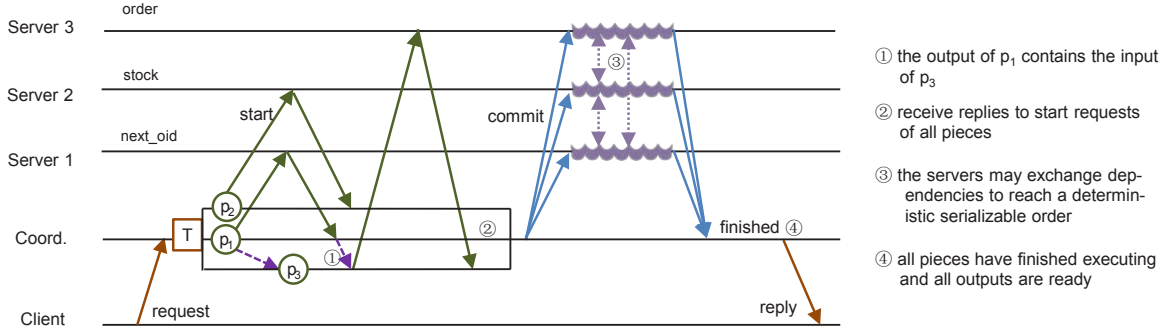
Figure 5: A typical ROCOCO message flow.

① the output of p_1 contains the input of p_3

② receive replies to start requests of all pieces

③ the servers may exchange dependencies to reach a deterministic serializable order

④ all pieces have finished executing and all outputs are ready

```
function Server S::start_req(p):
  S.dep[p.owner].status = STARTED
  foreach p' received by S that conflicts with p
    if p.immediate == true
      add p'.owner →ⁱ p.owner to S.dep
    else
      add p'.owner →ᵈ p.owner to S.dep
  if p.immediate == true
    output = execute(p)
  return (S.dep, output)
```

Figure 6: How a server processes a start request.

ply aggregates the returned dependency graphs from all involved servers (not shown in psuedocode).

The commit phase. The commit phase of a transaction distributes dependency information for all pieces, ensures each server can safely decide if a piece must be reordered, deterministically reorders pieces on each server if necessary, executes deferred pieces, and commits a transaction.

The coordinator begins the commit phase once it has sent out all start requests and collected their responses. For each participating server, the coordinator sends a commit request containing the aggregated dep graph. When aggregating a set of dependency graphs, one takes the union of vertexes/edges and sets each vertex's status to be the highest one in those graphs.

Figure 7 summarizes how a server handles a commit request in psuedocode. Upon receiving a commit request for transaction T, server S updates the status of T to COMMITTING in its dependency graph, if $T.status$ is lower than COMMITTING. Server S also aggregates the dependency information in the commit request into $S.dep$.

Next, server S ensures it can safely decide if its piece of T should be reordered by collecting the transitive closure of T's conflicting transactions' in $S.dep$. To do this, it examines $S.dep$ to find all T' that are ancestors of T and waits for the status of those T' to become COMMITTING or DECIDED. In the common case when T' involves

server S and S will eventually receive the commit request of T' so it simply waits; in the uncommon case when T' does not involve S, it issues a status request for T' to a server S' involved in T'. S' replies with its dependency graph after S' has received the commit request of T'. S aggregates the received graph with its own.

Next, server S calculates the strongly connected component (SCC) of T in $S.dep$, denoted T^{SCC}, which typically includes only T.[2] The server then sets the status of all transactions in T^{SCC} to DECIDED. Next, the server waits for all ancestors of the T^{SCC} to become DECIDED. Furthermore, for each ancestor T' involving server S, S also waits for $T'.finished$ to become true.

Next, to decide the right execution order for T, server S topologically sorts T^{SCC} according to $\xrightarrow{i}$ edges. To ensure that different servers reach a single sorting order, sorting is done deterministically. This topological sort is possible if and only if there are no cycles in T^{SCC} connected by only $\xrightarrow{i}$ edges. ROCOCO's offline checker ensures this will always be the case by eliminating any SC-cycle whose C-edges all have the I-I type. We elaborate this argument further in § 3.3.

Finally, server S executes the deferred pieces of each transaction T in T^{SCC} that involves S in the sorted order. Upon finishing executing T, server S sets $T.finished$ to be true and returns the results to T's coordinator.

When a coordinator has collected the responses from all participating servers, the transaction is considered committed and the output is returned to the client.

3.3 Correctness

This subsection presents a proof sketch of correctness. A more rigorous version of the proof is available in a technical report [47]. Specifically, we prove that ROCOCO

---

[2] We use the Tarjan algorithm [53] for SCC computation. In the best case, only those nodes and edges in the SCC need to be visited; In the worst case, all nodes and edges in the graph need to be visited and the complexity is $O(V+E)$.

```
function Server S::commit_req(T, dep):
  S.dep ⊎= dep
  S.dep[T].status ⊎= COMMITTING
  foreach T' ↝ T in dep
    if T' does not involve S and
        S.dep[T'].status == STARTED
      contact S' involved in T' and
        wait until S.dep[T'].status ≥ COMMITTING
  T^SCC = find_SCC(T, S.dep)
  foreach T' not in T^SCC and T' ↝ T^SCC
    wait until S.dep[T'].status == DECIDED
    if T' involves S
      wait until T'.finished == true
  deterministic_topological_sort(T^SCC)
  foreach T' in T^SCC  # including T
    S.dep[T'].status = DECIDED
    if T' involves S and
        T'.finished == false
      foreach deferred p' of T'
        p'.output = execute(p')
        T'.output ⊎= p'.output
        T'.finished = true
  return T.output
```

Figure 7: How a server processes a commit request.

guarantees strict serializability:

Serializability: [12] The committed transactions have an equivalent serial schedule, such that all conflicting operations in the actual schedule are ordered in the same way as in the equivalent serial schedule.

Strict-serializability: [12, 31] The above serial schedule preserves the real-time order, i.e., if transactions T_1 commits before T_2 starts in real time, T_1 appears before T_2 in the equivalent serial schedule.

The proof involves arguments on the *serialization graph*, which is a directed graph where each vertex represents a transaction and each edge represents an ordered conflict. Suppose transactions T_1 and T_2 have conflicting accesses (at least one is a write) to the same data item x. If T_1 accesses x before T_2 does, the serialization graph contains a $T_1{\rightarrow}T_2$. To prove ROCOCO is serializable, we must show that any serialization graph it generates is acyclic [13].

First, we show that all relations in the serialization graph are captured in the dependency information collected by servers.

Lemma 1. For any transactions T_1 and T_2, if $T_1{\rightarrow}T_2$ is in the serialization graph, then $T_1{\rightarrow}T_2$ must be included in the commit request of T_2.
Proof Sketch. By definition, $T_1{\rightarrow}T_2$ in the serialization graph implies that a pair of conflicting pieces, p_1 of T_1 and p_2 of T_2, exist and that p_1 executes before p_2 on a corresponding server S. Because the offline checker has eliminated all I-D conflicts, p_1 and p_2 are either 1) both immediate pieces, or 2) both deferrable pieces. In scenario 1), $T_1{\rightarrow}T_2$ in the serialization graph means p_1 executes before p_2 in the start phase and server S adds

$T_1{\rightarrow}T_2$ to $S.dep$. By the ROCOCO protocol, this dependency will be sent back to the coordinator, aggregated with other dependencies, and then appear in T_2's commit requests. In scenario 2), if p_1 executes before p_2, then T_1 has arrived before T_2 at some server S' in the start phase, resulting in $T_1{\rightarrow}T_2$ in $S'.dep$. Again, by the ROCOCO protocol, this dependency will be included in T_2's commit request.

Next, we argue that ROCOCO never generates a cycle in the serialization graph, due to a combination of servers breaking SCCs with deferred execution and the offline checker eliminating unreorderable SC-cycles.

Proposition 1. The serialization graph is acyclic.
Proof Sketch. For proof by contradiction, we assume there exists such a cycle (δ) of transactions in the serialization graph. First, we observe that each server involved in δ has δ in its dependency graph prior to executing any transaction in δ in the commit phase. The proof for this observation is in [47] and is based on Lemma 1 and the specification of ROCOCO that ensures each involved server transitively capture conflicting transactions in one SCC. Next, we note that the cycle δ must contain at least one pair of deferrable pieces. If all the pieces in δ are immediate, then δ corresponds to a SC-cycle involving only I-pieces, which would have been detected and eliminated by the offline checker. Last, we obtain a contradiction from the specification of ROCOCO that would have reordered the deferrable pieces to break δ.

Proposition 2. For any transactions T_1 and T_2, if T_2 starts after T_1 has finished, the serialization graph does not contain a path from T_2 to T_1, $T_2{\rightsquigarrow}T_1$.
Proof Sketch. To prove by contraction, we assume $T_2{\rightsquigarrow}T_1$ exists in the serialization graph. For any $T_i{\rightarrow}T_j$ in the serialization graph, in order for T_j to become COMMITTING on any server S, ROCOCO requires S to have waited for T_i to become COMMITTING. Therefore, given a path $T_2{\rightarrow}T_i{\rightarrow}T_j{\rightarrow}..{\rightarrow}T_1$ in the serialization graph, we can follow the path in reverse and deduce that T_2 has a status of COMMITTING at some server before T_1 becomes COMMITTING. This implies that T_2 has begun its commit phase before T_1 has finished at all servers, which contradicts the fact that T_2 has not started.

Proposition 1 implies serializability. Proposition 2 additionally shows strict-serializability.

3.4 Reducing Dependency Graphs

In the basic protocol, a server's dependency graph is verbose and grows without bound over time. We now explain how to more efficiently store and transmit dependency information.

To reduce the number of edges in $S.dep$, server S only adds *the nearest dependencies* of T in the graph upon

receiving T's start phase. The nearest dependencies of T have the longest path of one hop to T. However, in contrast to previous work that also tracked the nearest dependencies [42, 43], ROCOCO has two types of edges and paths. If a path contains at least one $\xrightarrow{i}$ edge, it is called an i-path. If a path consists of only $\xrightarrow{d}$ edges, it is called a d-path. An i-path is a stronger type than d-path. A path is longest only if there are no other longer paths with the same or a stronger type.
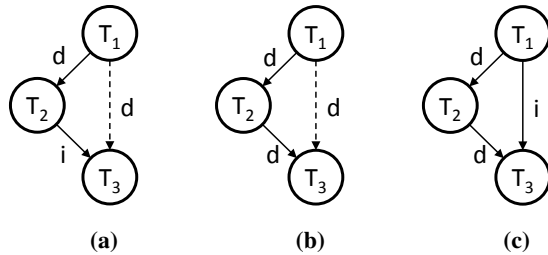

(a)   **(b)**   **(c)**

Figure 8: Nearest dependencies and longest paths are shown with solid arrows. $T_1\xrightarrow{d}T_3$ is not a longest path in the left and middle figures and thus is safely removable. $T_1\xrightarrow{i}T_3$ is a longest path in the right figure and cannot be omitted.

For example, suppose that server S has $T_1\xrightarrow{d}T_2$ in $S.dep$. When S receives a deferrable piece of T_3 that conflicts with both T_1 and T_2, instead of adding $\{T_1\xrightarrow{d}T_3, T_2\xrightarrow{d}T_3\}$ to $S.dep$, S only adds $\{T_2\xrightarrow{d}T_3\}$. Skipping $T_1\xrightarrow{d}T_3$ is acceptable because the dependency is still tracked by $T_1\xrightarrow{d}T_2$ and $T_2\xrightarrow{d}T_3$ (Figure 8a). As another example, suppose that $S.dep$ contains $\{T_1\xrightarrow{d}T_2\}$ and a new transaction T_3 attempts to add $\{T_1\xrightarrow{i}T_3, T_2\xrightarrow{d}T_3\}$. In this case, edge $T_1\xrightarrow{i}T_3$ cannot be skipped, because the path $T_1\xrightarrow{d}T_2\xrightarrow{d}T_3$ does not capture the stronger ordering constraint of $T_1\xrightarrow{i}T_3$ (Figure 8c).

In practice, ROCOCO tracks *one-hop dependencies*, a slightly larger superset of nearest dependencies. When server S receives a new piece p, it finds only the most recent conflicting piece p' for each of p's conflicts and adds $T'\rightarrow T$ to dep. Therefore, if the number of items a piece accesses is constant, then the time and space complexity of handling a new piece is $O(1)$.

In the basic protocol, server S returns the full graph $S.dep$ to the coordinator in the start phase. This is unnecessary. In particular, the coordinator only needs to learn of T's ancestors that are not yet DECIDED. Therefore, server S only computes the subgraph of $S.dep$ containing T's ancestors whose status is lower than DECIDED. Also, in its reply to a status request for T, a server only needs to include every undecided ancestors of T if T is not yet decided; if T is already DECIDED, the server replies with T^{SCC}.

A transaction is considered committed if the coordinator has received commit replies from all involved servers. It is tempting to simply remove all committed transactions from a server's *dep*. However, it is not correct to do so because the server may receive a status request for its committed transaction from another server. To garbage collect, ROCOCO uses an epoch mechanism similar to previous work [55, 32]. Each server keeps an epoch number that slowly increases. A transaction is tagged with an epoch number when it starts at a server. The epoch number on a server increases only after all transactions in the last epoch are all committed, and no other server falls behind or has ongoing transactions at one or more epochs ago. Dependencies from two epochs ago can be safely discarded.

3.5 Fault Tolerance

To tolerate failure, each server and the coordinator need to persist its transaction log to disks and preferably also replicates it across machines using a Paxos-based replication system [39, 15]. In ROCOCO, the coordinator logs the transaction request before starting the transaction, in case it fails during execution. Each server logs each start request following its arriving order, including its type and input. It does not need to log its output, because the output is deterministic once the order of start requests is fixed.

If a coordinator fails, after it recovers it will send the start requests again to all involved servers. For a server receiving the request, it first examines whether it has received this start request before. If so, it returns the same execution result and dependency graph; If not, it handles this request normally.

If a server fails, when it recovers it needs to replay all the start requests before it responds to other requests. In order to commit these transactions during recovery, the server asks other servers about the corresponding commit requests. In corner cases, such as all servers crashing, the servers should let the coordinator restart the affected transactions. To accelerate the recovery process, the server can also log when a transaction commits (i.e. its finished flag becomes true), but this is off the critical path of a transaction.

4 Extension

We describe two extensions to the basic design of § 3. § 4.1 shows how to optimize read-only transactions. § 4.2 explains how ROCOCO copes with merged pieces that internally rely on traditional distributed concurrency control.

```
function Coordinator C::do_ro_txn(T):
  # chop the transaction into pieces.
  # for a piece pᵢ, inputᵢ is its input,
  # sᵢ is its server
  foreach pᵢ in T
    wait until inputᵢ is ready
    outputᵢ = sᵢ.ro_req(T, pᵢ, inputᵢ)
  repeat
    # save the result of the last round read,
    # and issue another round.
    output' = output
    reset(input)
    foreach pᵢ in T
      wait until inputᵢ is ready
      outputᵢ = sᵢ.ro_req(T, pᵢ, input'ᵢ)
    # succeed if the two rounds return the same
  until output = output'
  return output
```

Figure 9: Coordinator read-only transaction

4.1 Read-Only Transactions

Read-only transactions often make up a significant fraction of OLTP workloads. Moreover, they often contain many immediate reads that increase the likelihood of SC-cycles without a D-D edge. To avoid this increase we provide a separate solution to execute read-only transactions that allows the offline checker to exclude read-only transactions from the constructed SC-graph.

To process a read-only transaction, the coordinator sends a round of read requests to each involved server. When a server receives the request it waits for all conflicting transactions to become FINISHED and then it executes the read and returns the result. After the coordinator finishes this round, it issues a second round of requests which are identical to the first round, i.e., they also wait for all conflicting transactions to become FINISHED. The transaction is considered successful if both rounds return the same results. If the results do not match, the coordinator simply re-starts the transaction.

Waiting for all conflicting transaction to finish is the key to ensuring the combination of this read-only transaction algorithm with the rest of ROCOCO is still strictly serializable. When one server waits for a transaction to finish it is forcing that transaction to at least start on all other involved servers. Then, if a first round read happened on a different server before that transaction, its corresponding second round read will at least encounter the transaction in the start phase. The second round read will wait for it to finish before executing, which ensures it will see a different result from the first round and force another round of reads.

4.2 Merged Pieces

In § 3, we assume that the offline checker finds only reorderable SC-cycles so that each piece only involves one

```
function Server S::ro_req(T, p, input):
  foreach T' in S.dep and T' involves S
      and T' conflicts with piece p
    wait until T'.finished is true
  output = execute(p)
  return output
```

Figure 10: Server read-only transaction

server at runtime. When the offline checker discovers unreorderable SC-cycles, it combines the pieces in the cycle that belong to the same transaction into a single *merged piece*. In contrast with the simple pieces discussed above that execute on a single server, a merged piece can be distributed across multiple servers. ROCOCO relies on traditional distributed concurrency control to execute each merged piece atomically across servers.

Fortunately, merged pieces are simple to integrate into the overall design of ROCOCO. A merged piece contains only immediate simple pieces, otherwise, it would not have contributed to an unreorderable SC-cycle. This allows the coordinator to use an OCC-based protocol to execute the sub-pieces of a merged piece in the start phase. Each server returns its dependency information in the normal way.

For example, suppose piece p_2 in Figure 4 is an immediate piece. As a result, p_1 and p_2 and their counterparts in the other new-order instance lead to an unreorderable SC-cycle. To eliminate this unreorderable SC-cycle, ROCOCO must execute p_1 and p_2 as a single merged piece. In the start phase, the coordinator executes p_1 and p_2 using a three-phase OCC+2PC (execute-prepare-commit). If OCC+2PC aborts the coordinator retries until it succeeds. In the commit phase of OCC+2PC, each server will then add appropriate edges and vertexes to its *dep* graph, and reply with all undecided ancestor transactions in *dep*, as in the normal start phase of ROCOCO.

In our experience, simple workloads such as RUBiS[3] and Retwis[2] do not require merged pieces. TPC-C is much more complex. However, with the support of read-only transactions, there are no unreorderable SC-cycles in TPC-C and therefore no merged pieces.

5 Evaluation

Our evaluation explores two key questions:

1. How does the throughput and latency of ROCOCO compare to that of traditional approaches under varying levels of contention?
2. Can ROCOCO scale out with OLTP workloads?

This section will show that ROCOCO has higher throughput and lower latency than OCC and 2PL under all levels of contention and that as contention increases

ROCOCO's advantage increases. It will also show that ROCOCO scales near linearly in a complex workload, where contention rate grows as the system scales.

5.1 Implementation

We implemented a distributed in-memory key-value store with transactional support using ROCOCO. Our prototype contains over 20 000 lines of C++ code, of which 10 000 are for concurrency control. It uses a custom RPC library implemented by one of the authors for communication [4]. It adopts the simple threading model of H-Store [52] that uses a single worker thread on each server (core) to sequentially process the server's transaction pieces. The worker thread performs all blocking operations asynchronously. Currently, stored procedure—i.e., a piece of a transaction—is written as a C++ function that is loaded into the server binary at launch time.

2PL and OCC implementation. Our prototype also implements 2PL+2PC or OCC+2PC. Both protocols include an execute phase in which the coordinator instructs each involved server to execute a transaction piece and then a commit phase based on 2PC.

For 2PL, servers acquire locks during the execute phase. Subsequently, in the 2PC prepare phase, the coordinator instructs each involved server to durably log its buffered writes and lock acquisitions. In 2PC's commit phase, servers release locks and make writes visible. We use the wound-wait strategy [48], also used in Spanner[19], to avoid deadlocks.

For OCC, servers return the versions of data items read to the coordinator during the execute phase. In 2PC's prepare phase, each involved server acquires write locks, acquires read locks to validate the freshness of reads, and durably logs its writes and vote decisions. In 2PC's commit phase, servers release locks and make writes visible.

5.2 Experimental Setup

Unless otherwise mentioned, all experiments are conducted on the Kodiak testbed [1]. Each machine has a single-core 2.6GHz AMD Opteron 252 CPU with 8GB RAM and Gigabit Ethernet. Most experiments are bottlenecked on the server CPU. We have achieved much higher throughput when running on a local testbed with faster CPUs.

In all experiments, clients and servers run on different machines. Each client machine runs 1-30 single-threaded client processes while each server machine runs a single server process. Each data point in the graphs represents the median of at least five trials. Each trial is run for over 60s with the first and last quarter of each trial elided to avoid start up and cool down artifacts.
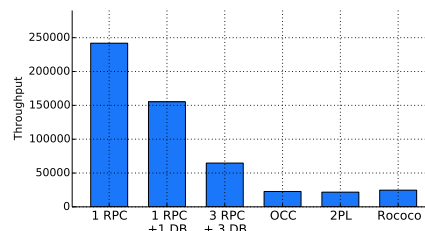

Figure 11: Throughput of baseline operations involving 3 servers. The transaction workload for OCC/2PL/ROCOCO has no contention.

Logging is turned off for all experiments because the Kodiak testbed does not include SSDs. We explore the overhead of logging to SSDs in our local testbed in Section 5.7. Logging always amplifies the throughput advantage of ROCOCO over 2PL and OCC. Logging sometimes increases the latency of ROCOCO over 2PL and OCC, but this is at most a few ms.

5.3 Micro-Benchmarks

To understand the base performance of our implementation, we ran a series of micro-benchmarks in a workload with *no contention*. The experiment uses three servers and its workload is a simple transaction that updates three counters, one on each server.

Figure 11 shows the throughput for a few baseline operations, from left to right, a null RPC to one of the servers (1 RPC), an RPC performing a database update at one of the servers (1 RPC+DB), three parallel RPCs each doing a database update at a different server (3 RPC+DB), and the simple transaction performing 3 database updates using OCC, 2PL, or ROCOCO.

Each server is able to handle ~75k null RPCs per second and is bottlenecked on CPU. The 1 RPC+DB throughput is slightly lower and is also bottlenecked on CPU, suggesting that the cost of a database access is relatively small compared to the cost of RPC. OCC and 2PL have similar throughput, roughly 1/3 of 3 RPC+DB, because they both require three rounds of RPCs to commit. ROCOCO requires two rounds of RPCs but incurs higher CPU cost to process dependency information, resulting in similar throughput to 2PL/OCC.

5.4 Scaled TPC-C Workload Overview

We evaluate ROCOCO's performance under contention using a scaled out version of the popular TPC-C [5] benchmark. This subsection explains how we scaled out TPC-C and how this differs from prior work.

Partition Strategy. Prior work partitioned the TPC-C database by warehouse [20, 54, 33, 55], with each
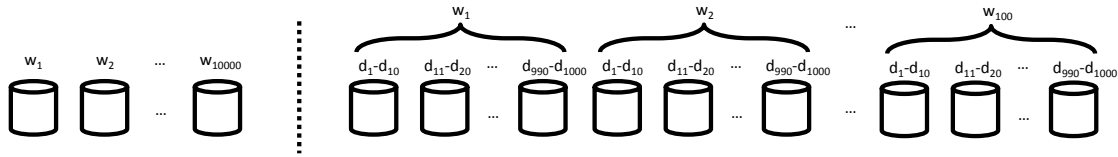
Figure 12: TPC-C sharding and scaling strategy. The left figure is the conventional strategy of scaling by increasing the number of warehouses. The right figure is our strategy of scaling inside a warehouse. w stands for warehouse, d stands for district.

server holding all data (including stock level and customer orders) related to a warehouse. This partition-by-warehouse strategy has two downsides. First, because only a single server handles each warehouse's data and requests, there is no performance scaling within a warehouse. This is acceptable in stock TPC-C that dictates a relatively low customer-to-warehouse ratio of 30K:1. However, in practice, a warehouse might need to handle a much larger population of users. For example, Amazon has >300 million customers served by ~100 warehouses. In these scenarios, the throughput of a single warehouse must scale beyond a single machine. Second, partition-by-warehouse does not stress the performance of distributed transactions, because only a minority ($<$15%) of all transactions involve more than one server.

To allow scaling within a warehouse, we partition the database by item or by district, of which there are many within a warehouse. Tables storing district related information are sharded according to `warehouse_id` and `district_id` (Figure 12). The stock level table is sharded by `warehouse_id` and `item_id`. We remove the `w_ytd` field that keeps track of the total value of purchases within a warehouse. To obtain the same information, we use a read-only transaction that reads the `d_ytd` value for each district and sums them up. This strategy avoids `w_ytd` from becoming a bottleneck for all new-orders within a warehouse. The original TPC-C benchmark uses a ratio of 30K:10:1 between customer, district and warehouse. We change it to 3M:1K:1 so that our ratio of customer-to-district remains the same as in the original TPC-C.
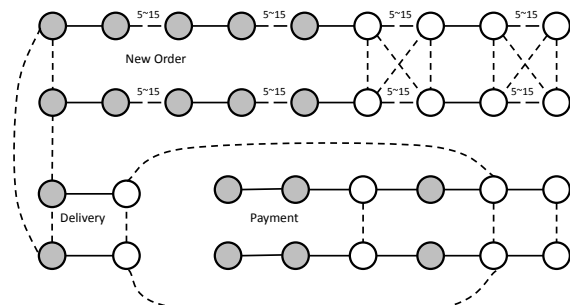

Figure 13: SC-graph for the TPC-C benchmark

Transaction pieces and the SC-graph. The TPC-C benchmark consists of five transactions: new_order, payment, delivery, order_status, and stock_level. Order_status, and stock_level are read-only transactions. Figure 13 shows the SC-graph for the remaining three transactions. The new order transaction contains five kinds of pieces, four of which occur 5 to 15 times, depending on how many items the transaction touches.

Table 1 shows the percentage of each transaction type in a random trial with ROCOCO, which matches the specified mix for TPC-C, and the average number of pieces included in each transaction.

	new-order	payment	order-status	delivery	stock-level
type	rw	rw	ro	rw	ro
ratio	44.97%	43.00%	4.03%	4.00%	4.00%
# pieces	40.97	4	3	40	210.93

Table 1: TPC-C commit transaction mix ratio in a ROCOCO trial. rw stands for general read-write transactions and ro stands for read-only.

5.5 Contention

We ran the scaled TPC-C benchmark to explore how 2PL, OCC, and ROCOCO perform under varying levels of contention. Figure 14 shows the results of this experiment. Figure 14a shows the throughput; Figure 14b shows the median, 90th percentile, and 99th percentile latency; and Figure 14c shows the commit rate.

Experimental Parameters. We ran the contention experiment with 8 servers that each served 10 districts. All 80 districts belong to 1 warehouse. We vary the clients per server from 1 to 100 with each client issuing a mixture of TPC-C transactions according to the specification in a closed loop. When the number of clients is higher, there are more requests per server and thus higher contention.

The contention level is also affected by the number of districts per server. If a core serves too few districts—e.g., 1 district per server—OCC and 2PL are unable to saturate the server's CPU under low contention. This is because a core needs at least four clients to saturate its
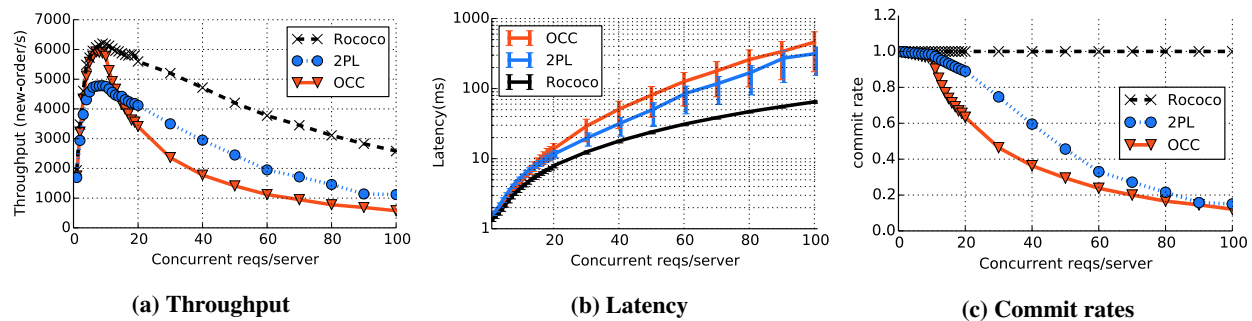
| (a) Throughput | (b) Latency | (c) Commit rates |

Figure 14: New-order transaction characteristics in TPC-C mixed workload benchmark, with 8 servers. In the latency graph, the line shows the 90th percentile, and bars show the median and 99th percentile.

CPU, but four clients on a single district causes many conflicts. If a core serves too many districts—e.g., 100 districts per server—a large number of clients are needed to generate even moderate levels of contention. In order to explore varying levels of contention, we configure each core to serve 10 districts.

Minimal Contention. When the number of concurrent requests per server is fewer than 10, there is almost no contention in the system. OCC reaches its maximum throughput, 5916 new-orders/s, with ~7 clients per server (Figure 14a) when each server's CPU is saturated. 2PL performs similarly, but has a lower maximum throughput, 4781 new-orders/s, because of the overhead of maintaining the read/write lock queues. ROCOCO has the highest maximum throughput, 6197 new-orders/s, because of computational savings from its one fewer round of RPCs, which outweighs the computational cost of the graph computations it performs.

Low Contention. When the number of concurrent requests increases from 10 to 20 per server, the contention level increases from minimal to low. OCC is very sensitive to this increase in contention with a large performance drop to only half of its peak throughput. This drop in throughput comes from repeated aborts and retries in OCC as evidenced by the drop in its commit rate to ~60%. 2PL is less sensitive to the increase in contention because it always allows the oldest transaction to commit, which guarantees progress and limits the number of retries for a transaction. This is also observable from the commit rate, which drops by only ~10%. The median latency of 2PL and OCC both increase to about 20ms, due to the abort/retry. The performance of RO-COCO is relatively unaffected because it does not abort on read-write transactions. The median latency of RO-COCO increase to 10ms, due primarily to more transaction requests waiting in the message queue.

Moderate Contention. When the number of concurrent requests increases from 20 to 40 per server, the contention increases from low to moderate. OCC con-

tinues to be very sensitive to this continued increase in contention. With 40 concurrent requests per server, the throughput of OCC is 1774 new-orders/s, one third of its peak, and its 99th percentile latency is over 67ms. The performance of 2PL also starts to drop quickly under moderate contention. Its throughput drops to 2950 new-orders/s, half of its peak, and its 99th percentile latency increases to 38ms. ROCOCO is also affected by the increase to moderate contention, though it is less sensitive than OCC and 2PL because it avoids aborting and retrying transactions. Its throughput drops by 24%, and its 99th percentile latency increases to 12ms.

High Contention. When the number of concurrent requests increases to over 40 per server, the benchmark reflects a high-level of contention. In the worst case, the throughput of OCC drops to a few hundred, due to large amounts of aborts and retries, with its commit rate dropping to 16%. 2PL has better performance than OCC, especially as measured by latency, because its wound-wait strategy ensures progress. But, 2PL's throughput and commit rate decrease significantly because of the large number of aborts. ROCOCO demonstrates the best performance with high contention. Its throughput drops to only 2584 new-orders/s, which is 130% higher than 2PL and 347% higher than OCC. More importantly, because ROCOCO avoids aborting and retrying, its latency is only 10%-40% of that of OCC and 2PL.

5.6 Scalability

We evaluate the scalability of ROCOCO in two different ways. The first is conventional TPC-C scaling by increasing the number of warehouses with a fixed number of districts per warehouse. In this case all protocols scale linearly (not shown) because each added warehouse is almost entirely independent of the existing warehouses and the contention rate—i.e., how frequently different transactions interact—remains constant.

The second, and more representative, experiment is scaling inside a warehouse from 10 districts on 1 server
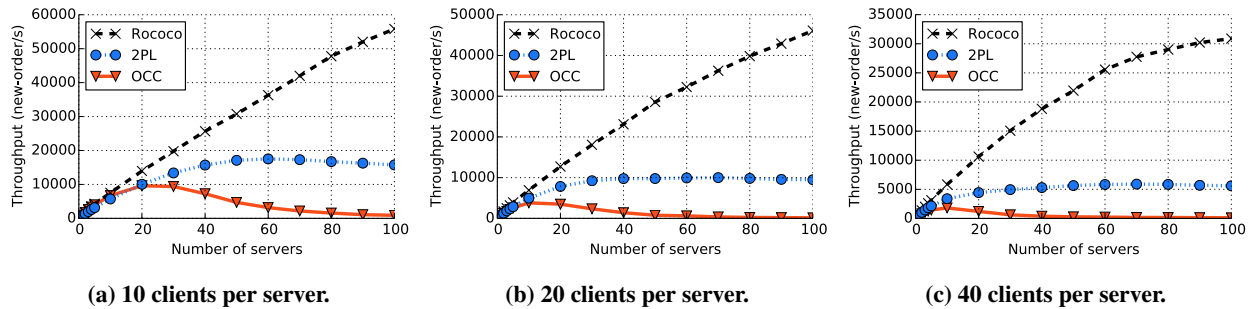
| (a) 10 clients per server. | (b) 20 clients per server. | (c) 40 clients per server. |

Figure 15: Scaling across districts

to 1000 districts on 100 servers. Scaling the number of districts increases the contention rate, which we believe is meaningful because as a system scale it is more likely that transactions will interact. For instance, in an e-commerce site, as the site becomes more popular it is likely to gain many more new customers than new items. In addition, a small set of items tend to be very popular and becomes more and more likely than have different customer try to concurrently purchase them, which is a source of contention in TPC-C new order transactions.

Our experiments use three levels of contention; low contention with 10 clients per server, shown in Figure 15a; moderate contention with 20 clients per server, shown in Figure 15b; and high contention with 40 clients per server, shown in Figure 15c.

ROCOCO scales near linearly when contention is low, with its throughput increasing from 7519 new-orders/s with 10 servers, to 13971 with 20 server, 25671 with 40 servers, and 47787 with 80 servers. The throughput of OCC and 2PL are far lower. OCC peaks at 9611 new-orders/s with 20 servers and 2PL peaks at 17521 with 60 servers. OCC and 2PL do not scale well because the increasing contention rate leads to more aborts.

ROCOCO also scales near linearly when contention is moderate, with its throughput increasing from 6921 new-orders/s with 10 servers, to 12736 with 20 server, 23117 with 40 servers, and 39853 with 80 servers. The higher levels of contention quickly lead to high abort rate for OCC and 2PL, which peak at 3816 and 10005 new-orders/s respectively.

When contention is high with 40 clients per server, ROCOCO still scales well, though this scaling is no longer near linear. The scaling is not linear because at this high level of contention ROCOCO propagates and processes much larger dependency graphs.

5.7 Logging to SSDs

This subsection explores the effect of synchronous logging in 2PL, OCC, and ROCOCO. This experiment is conducted in our local cluster that is equipped with SSDs, all other experiments were performed on the Ko-

diak cluster. To ensure that the log is safely persisted we turned off caching in the operating system and disks. We call `fsync` and wait for its return before we consider the log to be successfully written. We use a batch time of ~1ms before each `fsync`, which increases throughput significantly at the cost of slightly higher latency.

Table 2 shows the performance with 8 servers and 20 concurrent clients per server. 2PL and ROCOCO both have about 20% throughput drop and a latency increase of 2-3ms. The performance of OCC is more severely impacted as the batched logging resulting in requests holding their locks for longer in the prepare phase, which increases the likelihood of aborts. This effect is evident in the decreased commit rate for OCC.

	Throughput (new-orders/s)	Commit Rate (%)	Latency(ms) 50%	90%	99%
OCC					
no log	4109	63.82	8.49	11.35	13.60
logging	2748	54.28	12.17	18.35	22.79
2PL					
no log	4944	88.52	8.63	10.20	11.29
logging	4038	88.76	10.89	13.01	14.48
ROCOCO					
no log	6464	100	6.52	7.12	7.33
logging	5382	100	8.78	9.62	9.94

Table 2: Effect of logging in our local cluster

6 Related Work

General transactions with 2PL and OCC. Many seminal distributed databases such as Gamma [22], Bubba [16], and R[45] use forms of 2PL. Spanner [19] is Google's linearizable global-scale database that uses 2PL for read-write transactions and a separate timestamp based protocol from read-only transactions. Replicated Commit optimizes the across site latency in Spanner's commit protocol [44].

OCC is also used in several recent systems, such as H-Store [33] and VoltDB [6]. MDCC [35] uses OCC for geo-replicated storage. Percolator uses OCC to pro-

vide snapshot isolation [14]. Adya et al. [7] use loosely synchronized clocks and timestamps in the validation of OCC.

Observations have been made that OCC and 2PL behave well with no contention, but the performance will drop quickly as contention increases [8, 30]. This is also what we have observed in our evaluation.

Concurrency control with limited transactions. A recent trend is to improve performance by supporting limited types of distributed transactions. For example, MegaStore [10] only provides serializable transactions within a data partition. Other systems, such as Granola [20], Calvin [54] and Sinfonia [9] propose concurrency control protocols for transactions with known read-/write keys.

Sinfonia's protocol is based on OCC and 2PC. Granola achieves a deterministic serial order of conflicting transactions by exchanging timestamp between servers while Calvin achieves this by using a separate sequencing layer that assigns all transactions to a deterministic locking order to ensure isolation at each participating server. None of these systems supports key-dependent transactions: the read/write sets must be known apriori. In contrast, ROCOCO does support such transactions with immediate pieces.

Dependency and interference. Our work is motivated by recent efforts on consensus protocols such as Generalized-Paxos [38] and EPaxos [46], which uses dependency to reorder interfering commands in state machine replication (SMR). Paxos addresses consistent data replication and is used as a black-box module to provide replication in databases. However, consensus protocols bear some resemblance to distributed transaction protocols because reaching consensus is similar to committing a write-only transactions between several replicas of the same item [29].

COPS/Eiger [42, 43] track dependency between operations to provide causal+ consistency in geo-replicated key-value stores. Dependencies are also used to provide read-only/write-only transaction support. Warp [24] is a transaction layer on top of HyperDex [23] and its protocol also performs dependency tracking.

A major difference between ROCOCO and the above dependency-tracking systems is that ROCOCO can avoid aborts for transactions that require intermediate results between pieces. ROCOCO pushes this boundary using offline checking to eliminate possible unreorderable interleavings and by tracking finer grained dependencies to break dependency cycles in a serializable way.

Transaction decomposition and offline checking. The database community has explored various aspects of decomposing a transaction into smaller pieces for improved performance. [27, 11, 17, 25] Shasha et al. [50,

49] propose the theory of transaction chopping which uses SC-cycles to analyze possible conflicts that may lead to non-serializable execution. Lynx [57] uses transaction chopping and chain execution to achieve serializability and low latency simultaneously in a geo-distributed system. It uses commutative operations and origin ordering to ensure SC-cycles in web applications are safe. Compared to Lynx, ROCOCO distinguishes reorderable SC-cycles from unreorderable ones, executes pieces in parallel, and supports the strict form of serializability.

Geo-distributed systems with weaker semantics. Geo-distributed systems face a tradeoff between strong semantics and low latency. Systems such as Dynamo [21] and Cassandra [37] embrace latency and offer eventual consistency without transactional support. PNUTS [18] offers per-record timeline consistency. Walter provides parallel snapshot isolation [51] and Gemini provides Red/Blue consistency [40]. ROCOCO supports transactions with the strongest semantics (i.e. strict serializability) and thus will incur cross-datacenter latency when running in a geo-distributed setting.

7   Conclusion

This paper presented ROCOCO, a novel concurrency control protocol for distributed transactions. With the help of offline checking, ROCOCO reorders pieces of interfering transactions into a strict-serializable order and avoids aborts. In a scaled TPC-C benchmark ROCOCO outperformed conventional protocols and showed stable performance with increasing contention.

Acknowledgement

References

[1] Kodiak testbeds. http://portal.nmc-probe.org/.

[2] Retwis. http://retwis.antirez.com/.

[3] RUBiS. http://rubis.ow2.org/.

[4] Simple RPC in C++. https://github.com/santazhang/simple-rpc.

[5] TPC-C Benchmark. http://www.tpc.org/tpcc/.

[6] VoltDB. http://www.voltdb.com/.

[7] A. Adya, R. Gruber, B. Liskov, and U. Maheshwari. Efficient optimistic concurrency control using loosely synchronized clocks. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, SIGMOD '95, pages 23–34, New York, NY, USA, 1995. ACM.

[8] 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.

[9] M. K. Aguilera, A. Merchant, M. Shah, A. Veitch, and C. Karamanolis. Sinfonia: a new paradigm for building scalable distributed systems. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 159–174. ACM, 2007.

[10] J. Baker, C. Bond, J. Corbett, J. Furman, A. Khorlin, J. Larson, J.-M. Léon, Y. Li, A. Lloyd, and V. Yushprakh. Megastore: Providing Scalable, Highly Available Storage for Interactive Services. In *CIDR*, volume 11, pages 223–234, 2011.

[11] A. J. Bernstein, D. S. Gerstl, and P. M. Lewis. Concurrency control for step-decomposed transactions. *Information Systems*, 24(8):673–698, 1999.

[12] P. A. Bernstein and N. Goodman. Concurrency control in distributed database systems. *ACM Computing Surveys (CSUR)*, 13(2):185–221, 1981.

[13] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency control and recovery in database systems*, volume 370. Addison-wesley New York, 1987.

[14] P. Bhatotia, A. Wieder, İ. E. Akkuş, R. Rodrigues, and U. A. Acar. Large-scale incremental data processing with change propagation. In *Proceedings of the 3rd USENIX conference on Hot topics in cloud computing*, pages 18–18. USENIX Association, 2011.

[15] W. J. Bolosky, D. Bradshaw, R. B. Haagens, N. P. Kusters, and P. Li. Paxos replicated state machines as the basis of a high-performance data store. In *Proceedings of the 8th USENIX conference on Networked systems design and implementation*, pages 11–11. USENIX Association, 2011.

[16] H. Boral, W. Alexander, L. Clay, G. Copeland, S. Danforth, M. Franklin, B. Hart, M. Smith, and P. Valduriez. Prototyping Bubba, a highly parallel database system. *Knowledge and Data Engineering, IEEE Transactions on*, 2(1):4–24, 1990.

[17] Y. Breitbart, H. Garcia-Molina, and A. Silberschatz. Overview of multidatabase transaction management. *The VLDB Journal*, 1(2):181–239, 1992.

[18] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. PNUTS: Yahoo!'s hosted data serving platform. *Proceedings of the VLDB Endowment*, 1(2):1277–1288, 2008.

[19] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, et al. Spanner: Googles globally distributed database. *ACM Transactions on Computer Systems (TOCS)*, 31(3):8, 2013.

[20] J. Cowling and B. Liskov. Granola: low-overhead distributed transaction coordination. In *Proceedings of the 2012 USENIX conference on Annual Technical Conference*, pages 21–21. USENIX Association, 2012.

[21] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 205–220, New York, NY, USA, 2007. ACM.

[22] D. J. DeWitt, S. Ghandeharizadeh, D. A. Schneider, A. Bricker, H.-I. Hsiao, and R. Rasmussen. The Gamma database machine project. *Knowledge and Data Engineering, IEEE Transactions on*, 2(1):44–62, 1990.

[23] R. Escriva, B. Wong, and E. G. Sirer. Hyperdex: A distributed, searchable key-value store. In *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '12, pages 25–36, New York, NY, USA, 2012. ACM.

[24] R. Escriva, B. Wong, and E. G. Sirer. Warp: Lightweight Multi-Key Transactions for Key-Value Stores. Technical report, 2014.

[25] H. Garcia-Molina. Using semantic knowledge for transaction processing in a distributed database. *ACM Transactions on Database Systems (TODS)*, 8(2):186–213, 1983.

[26] H. Garcia-Molina, R. J. Lipton, and J. Valdes. A massive memory machine. *Computers, IEEE Transactions on*, 100(5):391–399, 1984.

[27] H. Garcia-Molina and K. Salem. Sagas. In *Proceedings of the 1987 ACM SIGMOD International Conference on Management of Data*, SIGMOD '87, pages 249–259, New York, NY, USA, 1987. ACM.

[28] H. Garcia-Molina and K. Salem. Main memory database systems: An overview. *Knowledge and Data Engineering, IEEE Transactions on*, 4(6):509–516, 1992.

[29] J. Gray and L. Lamport. Consensus on transaction commit. *ACM Transactions on Database Systems (TODS)*, 31(1):133–160, 2006.

[30] S. Harizopoulos, D. J. Abadi, S. Madden, and M. Stonebraker. OLTP through the looking glass, and what we found there. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 981–992. ACM, 2008.

[31] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, 1990.

[32] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. ZooKeeper: wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX conference on USENIX annual technical conference*, volume 8, pages 11–11, 2010.

[33] E. P. Jones, D. J. Abadi, and S. Madden. Low over-

head concurrency control for partitioned main memory databases. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 603–614. ACM, 2010.

[34] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. Zdonik, E. P. Jones, S. Madden, M. Stonebraker, Y. Zhang, et al. H-store: a high-performance, distributed main memory transaction processing system. *Proceedings of the VLDB Endowment*, 1(2):1496–1499, 2008.

[35] T. Kraska, G. Pang, M. J. Franklin, S. Madden, and A. Fekete. MDCC: Multi-data center consistency. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 113–126. ACM, 2013.

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

[37] A. Lakshman and P. Malik. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, 2010.

[38] L. Lamport. Generalized consensus and paxos.

[39] L. Lamport. Paxos made simple. *ACM Sigact News*, 32(4):18–25, 2001.

[40] C. Li, D. Porto, A. Clement, J. Gehrke, N. Preguiça, and R. Rodrigues. Making geo-replicated systems fast as possible, consistent when necessary. In *Proceedings USENIX Symposium on Operating System Design and Implementation (OSDI)*, 2012.

[41] K. Li and J. F. Naughton. Multiprocessor main memory transaction processing. In *Proceedings of the first international symposium on Databases in parallel and distributed systems*, pages 177–187. IEEE Computer Society Press, 2000.

[42] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Don't settle for eventual: scalable causal consistency for wide-area storage with COPS. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 401–416. ACM, 2011.

[43] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Stronger semantics for low-latency geo-replicated storage. In *Symposium on Networked Systems Design and Implementation*, 2013.

[44] H. Mahmoud, F. Nawab, A. Pucher, D. Agrawal, and A. El Abbadi. Low-latency multi-datacenter databases using replicated commit. *Proceedings of the VLDB Endowment*, 6(9):661–672, 2013.

[45] C. Mohan, B. Lindsay, and R. Obermarck. Transaction management in the R* distributed database management system. *ACM Transactions on Database Systems (TODS)*, 11(4):378–396, 1986.

[46] I. Moraru, D. G. Andersen, and M. Kaminsky. There is more consensus in Egalitarian parliaments. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 358–372. ACM, 2013.

[47] S. Mu, Y. Cui, Y. Zhang, W. Lloyd, and J. Li. Extracting More Concurrency from Distribted Transactions. Technical Report TR2014-970, New York University, Courant Institute of Mathematical Sciences, 2014.

[48] D. J. Rosenkrantz, R. E. Stearns, and P. M. Lewis II. System level concurrency control for distributed database systems. *ACM Transactions on Database Systems (TODS)*, 3(2):178–198, 1978.

[49] D. Shasha, F. Llirbat, E. Simon, and P. Valduriez. Transaction chopping: Algorithms and performance studies. *ACM Transactions on Database Systems (TODS)*, 20(3):325–363, 1995.

[50] D. Shasha, E. Simon, and P. Valduriez. Simple rational guidance for chopping up transactions. In *ACM SIGMOD Record*, volume 21, pages 298–307. ACM, 1992.

[51] Y. Sovran, R. Power, M. K. Aguilera, and J. Li. Transactional storage for geo-replicated systems. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 385–400. ACM, 2011.

[52] M. Stonebraker, S. Madden, D. J. Abadi, S. Harizopoulos, N. Hachem, and P. Helland. The end of an architectural era:(it's time for a complete rewrite). In *Proceedings of the 33rd international conference on Very large data bases*, pages 1150–1160. VLDB Endowment, 2007.

[53] R. Tarjan. Depth-first search and linear graph algorithms. *SIAM journal on computing*, 1(2):146–160, 1972.

[54] 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*, pages 1–12. ACM, 2012.

[55] 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*, SOSP '13, pages 18–32, New York, NY, USA, 2013. ACM.

[56] A. Whitney, D. Shasha, and S. Apter. High volume transaction processing without concurrency control, two phase commit, sql or C++. In *Seventh International Workshop on High Performance Transaction Systems, Asilomar*, 1997.

[57] Y. Zhang, R. Power, S. Zhou, Y. Sovran, M. K. Aguilera, and J. Li. Transaction chains: achieving serializability with low latency in geo-distributed storage systems. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 276–291. ACM, 2013.