All in One: Design, Verification, and Implementation of SNOW-Optimal Read Atomic Transactions

SI LIU, ETH Zürich, Switzerland

Distributed read atomic transactions are important building blocks of modern cloud databases which magnificently bridge the gap between data availability and strong data consistency. The performance of their transactional reads is particularly critical to the overall system performance as many real-world database workloads are dominated by reads. Following the SNOW design principle for optimal reads, we develop LORA, a novel SNOW-optimal algorithm for distributed read atomic transactions. LORA completes its reads in exactly one round trip, even in the presence of conflicting writes, without imposing additional overhead to the communication and outperforms the state-of-the-art read atomic algorithms.

To guide LORA's development we present a rewriting-logic-based framework and toolkit for design, verification, implementation, and evaluation of distributed databases. Within the framework, we formalize LORA and mathematically prove its data consistency guarantees. We also apply automatic model checking and statistical verification to validate our proofs and to estimate LORA's performance. We additionally generate from the formal model a correct-by-construction distributed implementation for testing and performance evaluation under realistic deployments. Our design-level and implementation-based experimental results are consistent, which together demonstrate LORA's promising data consistency and performance achievement.

CCS Concepts: • **Software and its engineering → Formal methods**; **Development frameworks and environments**; • **Information systems → Distributed database transactions**; **Database performance evaluation**.

Additional Key Words and Phrases: model checking, statistical verification, data consistency, the SNOW theorem

1 INTRODUCTION

Modern web applications are typically layered on top of a high-performance cloud database running in a partitioned, geo-replicated environment for system scalability, data availability, and fault tolerance. As network partitions/failures (P) are unavoidable in general, a cloud database design must sacrifice either strong data consistency (C) or high data availability (A) according to the CAP theorem [17]. To balance well the trade-off between C and A, there is therefore a plethora of data consistency models for distributed database systems, from weak models such as *read committed* through various forms of *snapshot isolation* to the strongest guarantee *strict serializability*.

Read atomicity (RA) [10]—either all or none of a transaction's updates are visible to another transaction's reads—magnificently bridges the gap between C and A in a distributed setting by providing the strongest data consistency that is achievable with high availability (the HAT semantics [9]). Many industrial and academic distributed databases have therefore integrated *read atomic* transactions as important building blocks [4, 10, 21, 48, 53, 75]. To cite a few examples, RAMP-TAO [21] has recently layered RA on Facebook's TAO data store [18] to provide atomically visible and highly available transactions. The ROLA transaction system [48] implements a new stronger data consistency model, *update atomicity*, by extending RA with the mechanisms for preventing the lost update phenomena.

Many real-world database workloads are dominated by reads, e.g., in Facebook's TAO [18], LinkedIn's Espresso [67], and Google's Spanner [24], and thus the performance of transactional reads is critical to the overall system performance. Cloud database designers have dedicated tremendous efforts to the specific algorithms for transactional reads [10, 24, 25, 52, 53, 59] such as the efficient RAMP (Read Atomic Multi-Partition) algorithms [10].

Author's address: Si Liu, si.liu@inf.ethz.ch, ETH Zürich, Switzerland.

The SNOW theorem [54] is an impossibility result for transactional reads that proves no read can provide *strict serializability* (S) with non-blocking client-server communication (N) that completes in exactly one round trip and with only one returned version of the data (O) in a database system with concurrent transactional writes (W). The SNOW theorem is a powerful lens for determining whether transactional reads are optimal: for a distributed transaction algorithm designed for data consistency weaker than *strict serializability*, e.g., *read atomicity*, it must achieve the N+O+W properties to be SNOW-optimal. Nonetheless, the state-of-the-art *read atomic* algorithms are all SNOW-suboptimal (a more detailed overview of the suboptimality is given in Section 3.1): even the "fastest" algorithm RAMP-Fast [10] requires 1.5 round trips on average to complete its reads, thus violating the O property; one conjectured optimization for RAMP-Fast attempts to improve the overall database performance by sacrificing the *read your writes* (RYW) session guarantee [77] (a prevalent building block of many databases such as Facebook's TAO), which still incurs two round-trip times for reads in the presence of racing writes.

Our LORA Design. Challenges for deriving SNOW-optimal *read atomic* transactions are: (i) improving the already well-designed efficient *read atomic* algorithms such as RAMP-Fast; (ii) incurring no extra cost on transactional writes; and (iii) sacrificing no established consistency guarantee such as RYW. In particular, challenge (ii) may be more difficult: previous work [54] on deriving optimal transactional reads from the causally consistent data store COPS has been shown to provide suboptimal system performance [29]. The fundamental issue is that, to optimize the reads, the new design imposes additional processing costs on writes, which in turn reduce the overall throughput.

We present LORA, the *first* SNOW-optimal read atomic transaction algorithm. LORA's key idea is to compute an RA-consistent snapshot, represented by the timestamps of the requested data items, of the entire database for each transactional read which can then retrieve the corresponding RA-consistent version, indicated by the computed timestamp, in exactly one round trip. To do so, LORA uses a novel client-side data structure that maintains the view of the last seen timestamp for each data item, together with the associated sibling items updated by the same transaction. A RA-consistent snapshot can therefore be computed by comparing a data item's last seen timestamp and those timestamps for which it is a sibling item: returning the highest timestamp always guarantees RA.

LORA also employs the *two-phase commit protocol* as in existing read atomic algorithms to guarantee the atomicity of a transaction's updates. Differently, LORA decouples these two phases by allowing a transaction to commit after the first phase to improve the performance of transactional writes. To avoid loosing RYW as in the above conjectured optimization, LORA incorporates the committed writes into the view of the last seen timestamps in between the two phases for the subsequent RA-consistent snapshot computations. For the client-server communications in both phases, LORA imposes no more overhead than RAMP-Fast in terms of message payload for writes since the same metadata are piggybacked via each message.

In addition to optimizing read-only and write-only transactions (where transactional operations are only reads or writes), LORA also extends existing read atomic algorithms with the support for fully functional read-write transactions with mixed SNOW-optimal reads and RYW-assured one-phase writes.

From Design to Implementation. Developing correct and high-performance distributed database systems is hard with complex transactional read/write dynamics, especially in partitioned, geo-replicated environments. In particular, improving existing systems by touching a large code base often requires intensive manual efforts, has a high risk of introducing new bugs, and is not repeatable. In practice, very few optimizations can be explored in this way: in deriving SNOW-optimal algorithms, only one out of ten design candidates have been implemented and evaluated [54].

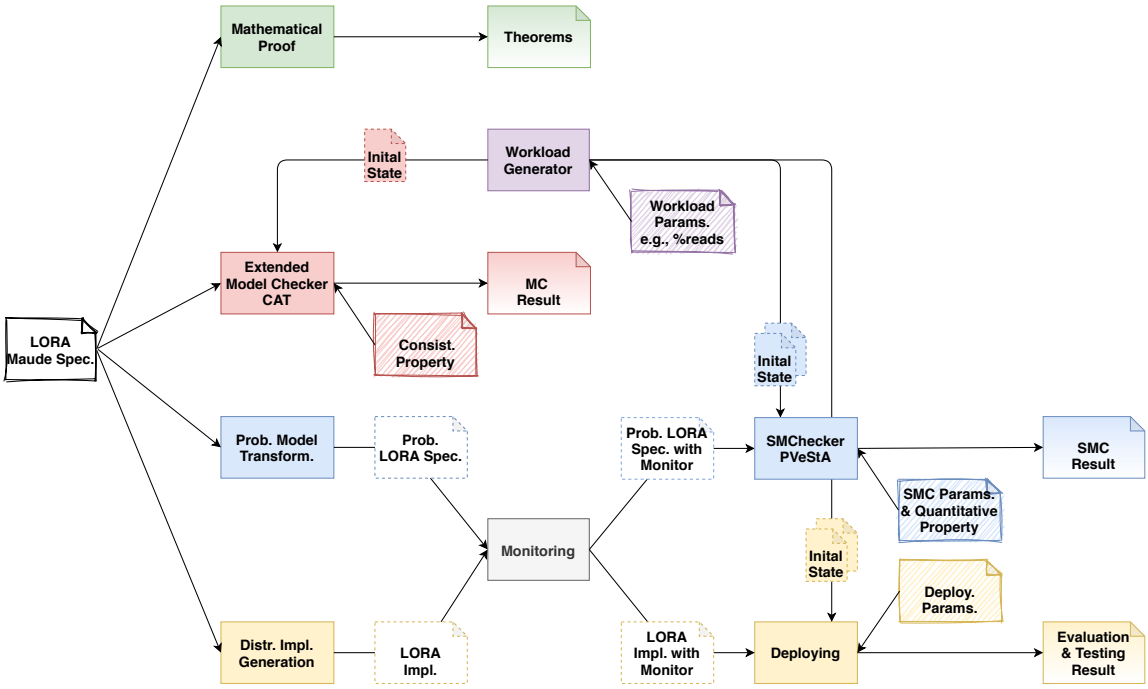

Fig. 1. The overview of our methodology and toolkit: The green, red, blue, and yellow parts refer to mathematical proofs of LORA's correctness, model checking of its consistency properties, SMC of its consistency/performance properties, and its implementation-based testing and evaluation under realistic deployment, respectively. The purple part is the parametric workload generator producing initial state(s) for the last three analyses. The gray part refers to the monitoring mechanism used to extract transaction measures. The solid boxes are mechanisms or tools. The inputs are sketched. The intermediate artifacts are dashed. The outputs are gradient.

Formal methods have been advocated and applied by both academia [6] and industry (e.g., Amazon Web Services [62] and Microsoft Azure [58]) to develop and verify cloud databases, particularly *at an early design stage* to establish both qualitative correctness (such as data consistency) and quantitative guarantees about the system performance (such as latency). To cite a few examples, Chapar [42] has been used to extract implementations of distributed key-value stores from their Coq-verified formal specifications; TLA+ and its model checker have been employed by Amazon for design-level verification; Microsoft Research has adopted the IronFleet framework [35] for verifying both system designs and implementations.

Our Methodology and Main Contributions. We use a combination of rewriting-logic-based techniques to guide the development of LORA (summarized in Fig. 1). Our methodology tackles *all* the following challenges that existing formal development methods fail to (see Section 9 for the detailed related work): (i) the complexity and heterogeneity of cloud database systems require a flexible and expressive formal framework [62]; (ii) complex data consistency properties necessitate the rigorous and automatic verification methods [62]; (iii) design-level quantitative assessments are equally important for cloud computing systems [6]; (iv) the pass from a verified design to a correct implementation is a necessity towards a production-ready system [76, 86]; and (v) the *semantic gaps* between different models (e.g., for correctness checking and for statistical verification) and between models and implementations must be bridged to obtain faithful analysis results [48, 51].

Regarding (i), we rely on rewriting logic's capability of modeling object-based distributed systems [57] and formalize our LORA design in the Maude language [22]. Such systems exhibit intrinsic features such as unbounded data structures and dynamic message creation that are quite hard or impossible to represent in other formalisms (e.g., finite automata).

To address (ii) we prove by induction that the formal model of LORA satisfies its claimed consistency properties RA and RYW. We also subject the model to the (extended) CAT tool [49], the state-of-the-art model checker for distributed transaction systems, for automatic model checking analysis of data consistency.

For challenge (iii), we quantitatively analyze LORA's transaction latency and throughput, as well as various consistency properties, via statistical model checking (SMC) [69, 88] that has demonstrated its predictive power in terms of system performance in an early design phase [44, 48]. Concretely, we first transform the nondeterminisitic, untimed Maude model of LORA (for the above qualitative analyses) into a purely probabilistic, timed model. We then add a monitoring mechanism to the transformed model that automatically extracts transaction measures during system executions. Finally, we formalize a number of data consistency and performance properties over such measures and apply Maude-based SMC with PVeStA [5] to statistically verify/measure LORA against the state-of-the-art *read atomic* algorithms. Our SMC results demonstrate LORA's promising performance and consistency achievement.

In addition to the above design-level formal analyses, we also perform the implementation-level testing and evaluation under realistic deployments, both to validate our claims from the qualitative analyses and to confirm our model-based SMC predications. In particular, we tackle challenge (iv) by using the *D* transformation [51] to generate a correct-by-construction distributed Maude implementation of LORA from its formal model. Such an implementation is further enriched by the monitoring mechanism for collecting runtime information before the automated deployment.

We bridge the semantic gaps by dedicating all the above formal efforts *within the same framework* and *with a single artifact*. To the best of our knowledge, this is the first demonstration that, within the same semantics framework, a new distributed transaction algorithm can be formally designed, rigorously verified, automatically analyzed for both qualitative and quantitative properties, correctly implemented, and evaluated under realistic deployments.

For fully automated analysis of extensive consistency properties and of system performance with both SMC and implementation-based evaluation, we provide additional contributions (also shown in Fig. 1) to the Maude ecosystem in the course of developing LORA: a workload generator for producing *realistic* database workloads; the extended CAT model checker for analyzing the RYW session guarantee; a monitoring mechanism for extracting transaction measures from both simulations and actual system runs; a library of performance and consistency metrics for quantitative analysis; and an automated tool for deploying distributed implementations in a cluster.

The remainder of this paper proceeds as follows: Section 2 gives preliminaries on distributed transactions, data consistency guarantees, and the SNOW theorem. Section 3 examines the suboptimality of existing *read atomic* algorithms and presents our LORA design. Section 4 gives an overview of the Maude ecosystem. Section 5 presents our formal specification of LORA in Maude, followed by the mathematical proofs in Section 6 and the automatic analysis in Section 7. Section 8 discusses the limitations and improvements to our methodology and toolkit. Finally, Section 9 discusses related work and Section 10 ends the paper with some concluding remarks and future work.

2 DISTRIBUTED TRANSACTIONS AND THE SNOW THEOREM

2.1 Distributed Transactions and Data Consistency

Today's web applications are layered atop a partitioned, geo-replicated distributed database. With data partitioning, very large amounts of data are divided into multiple smaller parts stored across multiple servers (or partitions) for

scalability. Replicating data in multiple distant physical sites improves data availability and system fault tolerance. A user application request is submitted to the database as a *transaction* that consists of a sequence of read and/or write operations[1] on data items (or keys) stored across multiple database partitions. Each transactional write creates a version of a data item, typically identified by a unique timestamp. A distributed transaction that enforces the ACID properties (Atomicity, Consistency, Isolation, and Durability) [13] terminates as either a committed transaction, signaling success, or an aborted one, signaling failure.

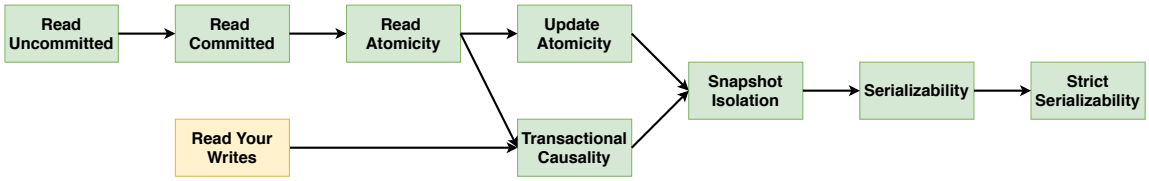

Fig. 2. Hierarchy of data consistency properties: $A \rightarrow B$ indicates that consistency property A is strictly weaker than consistency property B. Transactional and non-transactional consistency properties are colored in green and yellow, respectively.

Since network failures are unavoidable, a distributed database design must sacrifice either strong consistency or availability (the CAP theorem [17]). Providing strong data consistency guarantees for distributed transactions typically incurs high latency, e.g., a database system for *read atomicity* is expected to outperform an *update atomic* transaction system. There is therefore a spectrum of consistency models over partitioned replicated data for various web applications. Fig. 2 shows eight prevalent transactional consistency properties (in green) ranging from weak consistency such as *read uncommitted* to strong consistency like *strict serializability*:

- *Read uncommitted* (RU) [12] provides the weakest data consistency guarantee, where one transaction may see not-yet-committed changes made by other transactions.
- *Read committed* (RC) [12], used as the default data consistency model by almost all SQL databases, disallows a transaction from seeing any uncommitted or aborted data.
- *Read atomicity* (RA) [10] guarantees that either all or none of a distributed transaction's updates are *visible* to another transaction. Take a social networking application for example, if Thor and Hulk become "friends" in one transaction, then other transactions should not see that Thor is a friend of Hulk but that Hulk is not a friend of Thor; either both relationships are visible or neither is.
- *Update atomicity* (UA) [20, 48] strengthens *read atomicity* (see below) by preventing lost updates (a transaction's update to a data item is lost as it is overwritten by another transaction's update).
- *Transactional causality* (TC) [4, 53] ensures that a transaction reads from a snapshot of the data store that includes the effects of all transactions that causally precede it (*causal consistency*) and all transactional updates are made visible simultaneously, or none does (*read atomicity*).
- *Snapshot isolation* (SI) [12] requires a multi-partition transaction to read from a snapshot of a distributed data store that reflects a single commit order of transactions across sites, even if they are independent of each other.
- *Serializability* (SER) [66] ensures that the execution of concurrent transactions is equivalent to one where the transactions are run one at a time.

[1]A transaction that consists of only read, resp. write, operations is called a read-only, resp. write-only, transaction; otherwise, it is a read-write transaction with mixed read and write operations. Transactional reads refer to the read operations in both read-only and read-write transactions.

- *Strict serializability* (SSER) is the strongest consistency property that strengthens SER by enforcing the serial order to follow real time, thus providing the most aggressive data freshness (reads returning the latest writes).

In particular, we recall here RA's definition in [10]:

> "A system provides *read atomic* isolation if it prevents fractured reads anomalies and also prevents transactions from reading uncommitted, aborted, or intermediate data."

where (version i of data item x is denoted as x_i)

> "A transaction T_j exhibits *fractured reads* if transaction T_i writes versions x_m and y_n (in any order, with x possibly but not necessarily equal to y), T_j reads version x_m and version y_k, and $k < n$."

Note that timestamps are typically taken from a totally ordered set, thus inducing a total order on versions of each data item (e.g., $k < n$) and a partial order across versions of different data items.

Another useful class of data consistency guarantees refers to the client-centric ordering within a *session* that depicts a context persisting between transactions. For example, on a social networking website, all of a user's requests, submitted as transactions between login and logout, typically form a session. Terry *et al.* [77] proposed four session guarantees: *read your writes* (RYW), *monotonic reads* (MR), *writes follow reads* (WFR), and *monotonic writes* (MW). In this paper we consider RYW (colored in yellow in Fig. 2), the most prevalent session guarantee advocated by many industrial and academic database applications such as Facebook's TAO [18] and the RAMP transactions [10]. RYW guarantees that the effects of all writes performed in a (client) session are visible to its subsequent reads.

2.2 Snow-Optimality and Latency-Optimality

Many web applications today exhibit read-dominated database workloads [18, 24, 67]. Improving the performance of transactional reads has therefore become a key requirement for modern distributed databases. Lu *et al.* propose the SNOW theorem as a design principle for optimal transactional reads. The theorem states that it is impossible for a distributed transaction algorithm to achieve all four desirable properties (see Section 2.1 for *Strict serializability* (S)):

- *Non-blocking reads* (N) require that each server should process reads without blocking for any external event. Such operations are desirable as they can save the time that would be spent on blocking.
- *One response per read* (O) consists of two sub-properties: *one version per read* and *one round-trip to each server*. The *one version* sub-property requires that servers send only one value for each read as multiple versions would consume more time in data serialization/deserialization and transmission. The *one round-trip* sub-property requires the client to send at most one request to each server and the server to send at most one response back.
- *Write transactions that conflict* (W) requires that transactional reads that view data can coexist with conflicting transactions that are concurrently updating that data.

Definition 2.1 (SNOW-Optimality [54]). A distributed transaction algorithm is SNOW-optimal if its properties sit on the boundary of the SNOW theorem, i.e., achieving three out of the four SNOW properties.

Among the four properties, N+O particularly favor the performance of reads as *non-blocking* and *one response* lead to low read latency.

Definition 2.2 (Latency-Optimality [54]). A distributed transaction algorithm is latency-optimal if its reads are *non-blocking* (N) and it provides *one response per read* (O).

For a distributed transaction algorithm that is designed for data consistency weaker than *strict serializability* (e.g., *read atomicity* that we target in this paper), the most it can achieve is the N+O+W properties, i.e., SNOW-optimality. Such an algorithm is also latency-optimal (the N+O properties).[2]

3 THE LORA TRANSACTION ALGORITHM

In this section we first examine the suboptimality of existing read atomic designs, in particular the "fastest" algorithm RAMP-Fast (Section 3.1), and then present our SNOW-optimal read atomic transaction algorithm LORA (Section 3.2).

Assumptions. We make the same assumptions for LORA as in existing read atomic algorithms [10]: (i) a database is partitioned and each partition stores part of the entire database items (or keys); (ii) each key has a single logical copy, i.e., no data replication; (iii) we consider *multiversion concurrency control* [13] that allows reads to access not-yet-committed data; (iv) a timestamp uniquely identifies a version of a key; and (v) a transaction execution, initiated by a client (or coordinator), terminates in either commit or abort.

3.1 Suboptimality of Existing Read Atomic Algorithms

Three *read atomic* algorithms, RAMP-Fast (or RF), RAMP-Small (or RS), and RAMP-Hybrid (or RH), are presented in [10], offering different trade-offs between the size of message payload and the system performance in terms of the number of round trips required by a read to fetch the RA-consistent data. Two RAMP optimizations, Faster Commit (FC) and One-Phase Writes (1PW), are also conjectured by the original developers to reduce the overhead of processing reads and writes, respectively. Recently, the RAMP-TAO [21] protocol layers RAMP transactions on top of Facebook's TAO data store, providing replicated *read atomic* transactions.

None of these algorithms and conjectures are SNOW-optimal or latency-optimal. More specifically, RF needs 1.5 round trips on average to complete its reads (two round trips with conflicting writes and one round trip otherwise) and thus violates the O property; RS always requires two round trips for reads; RH may require two round trips even with no race between reads and writes; the FC design conjecture can only reduce the possibility of second-round reads in RF, thus not guaranteeing one round-trip reads in the presence of racing writes; the 1PW conjecture only optimizes transactional writes by sacrificing the *read your writes* session guarantee (see Appendix B for an example scenario and Section 7.3.2 for its formal analysis). RAMP-TAO inherits the suboptimality from the RAMP transaction algorithms.

In the following we recall the "fastest" algorithm RF (on which our LORA algorithm is built) and illustrate its suboptimality using a scenario of two concurrently executing transactions.

RAMP-Fast. To guarantee that all database partitions perform a transaction successfully or that none do, RF performs two-phase writes by using the *two-phase commit protocol*. The first phase initiates a PREPARE operation of a version $\langle key, value, ts, md \rangle$ on the partition storing *key*, where *ts* is the timestamp uniquely identifying this version and *md* is the metadata containing all other (sibling) keys updated in the same transaction. The second phase commits the transaction (via the COMMIT operations) if all involved partitions agree to (indicated by all PREPARED responses), where each partition updates an index containing the latest committed version of each key.

Regarding transactional reads, for each key RF first requests the highest-timestamped committed version stored on the partition (via a GET operation). By examining the returned timestamps and metadata, if RF finds a key in the

[2]In general, a SNOW-optimal algorithm is *not* necessarily latency-optimal, e.g., RIFL [40] satisfies the S+N+W properties; a latency-optimal algorithm may be SNOW-*suboptimal*, e.g., COPS-SNOW [54] that provides causal consistency is incompatible with write transactions.

metadata that has a higher timestamp *ts* than the returned timestamp, a second-round read (via GET) is issued to request the *ts*-stamped version. Once all such "missing" versions have been fetched, RF commits the transaction.

The pseudo-code of RF is shown in Appendix A where only read-only and write-only transactions (operations in a transaction are only reads, resp. writes) are considered.

Example 3.1 (Fig. 3-(a)). Consider a scenario of two conflicting transactions that read and update the same data items initialized as x_0 and y_0. Transaction T_1 performs two-phase writes on two partitions P_x and P_y that respectively store keys x and y. The concurrently executing transaction T_2 reads from P_x after P_x has committed T_1's write to x, while from P_y before P_y has committed T_1's write to y. Hence, T_2's first-round reads fetch the versions x_1 and y_0 (version i of key k denoted as k_i), the *fractured reads* that violate *read atomicity*. Using the metadata attached to its first-round reads, T_2 finds that y_1 is missing ($last[y] = 1$ and $1 > 0$), thus issuing a second-round read to fetch y_1 (indicated by the requested timestamp ts_{req}). The resulting versions x_1 and y_1 are therefore RA-consistent. T_2 is compatible with T_1 which can subsequently commits (y_1 on P_y).

In this example RF satisfies the N property (P_x and P_y return the requested versions without blocking), the W property (T_2's compatibility with T_1), and *read atomicity*, but *not* the O property (due to the second-round read to P_y). Hence, RF is neither SNOW-optimal nor latency-optimal.

3.2 The LORA Design

We present LORA, a SNOW-optimal read atomic transaction algorithm,[3] that optimizes and extends RF from three aspects: (i) providing one round-trip reads with *read atomicity*, (ii) committing writes in one phase with *read your writes*, and (iii) supporting read-write transactions. Algorithm 1 shows the pseudo-code of LORA with these major differences colored.

Regarding (i), LORA computes a RA-consistent snapshot of the database for each transactional read before it is issued out. Such a snapshot is indicated by the timestamps whose associated versions are subsequently returned by the database partitions. Hence, with a single round trip, transactional reads can fetch RA-consistent versions.

More specifically, the coordinator (or client) maintains a data structure $last[k]$ that maps each key k to its last seen timestamp *ts* and the associated metadata *md*, called sibling keys[4] (line 1, Algorithm 1). LORA invokes the GET_ALL method to request timestamped RA-consistent versions in parallel (lines 2-8): requesting k's version with the *maximum* of *ts* and all those timestamps for which k is a sibling key always prevents *fractured reads* (see Section 2.1), thus guaranteeing RA. Each partition invokes the GET method to return the requested timestamped value, together with the last committed timestamp for the requested key and the associated metadata used to update the coordinator's local view of $last[k]$ (lines 26-29).

Regarding (ii), LORA also employs the *two-phase commit protocol* as in RAMP-Fast to guarantee the atomicity of a transaction's updates. The PUT_ALL method uses a first round of communication to place each timestamped write on its respective partition (line 31-33) and a second round of communication to mark versions as committed (line 34-36). Unlike RAMP-Fast, LORA decouples these two phases by allowing the coordinator to return after issuing the PREPARE round (line 12-15) and to subsequently execute the COMMIT phase asynchronously (line 17-19). The coordinator updates the local view of $last[k]$ with its own writes in between the two phases (line 16) to ensure the *read your writes*

[3]Following the SNOW design principle, we do not strengthen the data consistency to *strict serializability* to achieve SNOW-optimality (S+N+W in this case) since doing so would change the base system into something new.

[4]Key x is called a sibling key of key y if both keys are written in the same transaction. For example, given transaction $T_i : [w(x_i), w(y_i), w(z_i)]$, the metadata associated to key x is the set of sibling keys $\{y, z\}$.

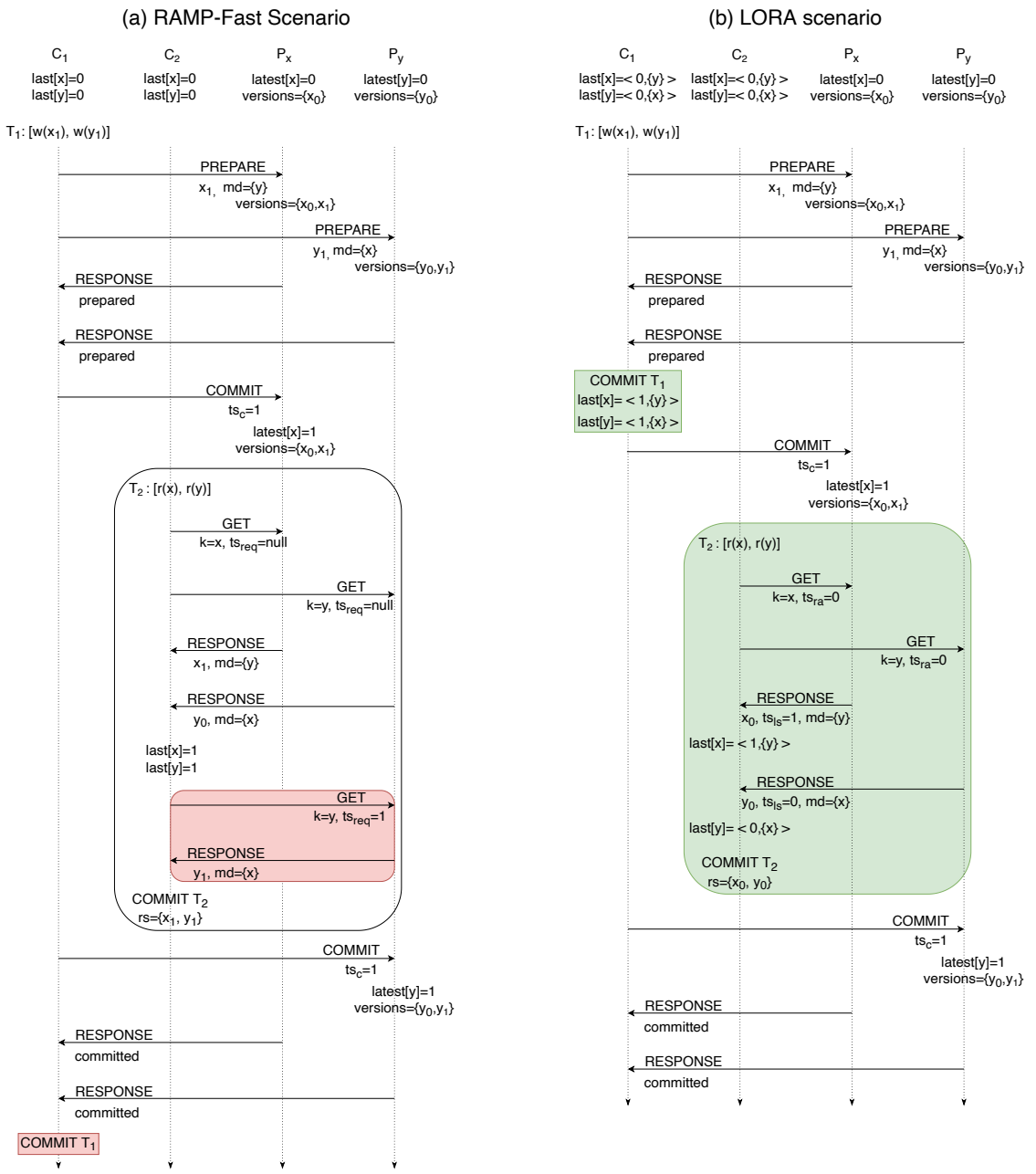

Fig. 3. Scenarios illustrating how LORA works differently from RAMP-Fast with conflicting reads and writes. The major differences in handling reads, resp. writes, are indicated by the shaded boxes.

Algorithm 1 LORA

Coordinator-side Data Structures

1: $last[k]$: last seen ⟨timestamp ts, metadata md⟩ for key k ▷ *only last seen timestamp for each key in RF*

Coordinator-side Methods

2: **procedure** GET_ALL(K : set of keys) ▷ *one round-trip read-only transactions*
3: read set $rs \leftarrow \emptyset$
4: **parallel-for** key $k \in K$ **do**
5: **for** key $k' : k \in last[k'].md$ **do** $ts_{ra} \leftarrow \max(last[k].ts, last[k'].ts)$ ▷ *RA-consistent snapshot*
6: $rs[k], last[k] \leftarrow$ GET(k,ts_{ra}) ▷ *update local view*
7: **end parallel-for**
8: **return** rs

9: **procedure** PUT_ALL(W : set of ⟨key, $value$⟩) ▷ *one-phase write-only transactions*
10: $ts_x \leftarrow$ generate new timestamp
11: $K_x \leftarrow \{w.key \mid w \in W\}$
12: **parallel-for** ⟨k, val⟩ $\in W$ **do**
13: $v \leftarrow \langle k, val, ts_x, K_x - \{k\}\rangle$
14: send PREPARE(v) to partition storing k
15: **end parallel-for**
16: **for** key $k \in K_x$ **do** $last[k] \leftarrow \langle ts_x, K_x - \{k\}\rangle$ ▷ *incorporate own writes*
17: **parallel-for** key $k \in K_x$ **do**
18: send COMMIT(ts_x) to partition storing k
19: **end parallel-for**

20: **procedure** UPDATE(K : set of keys, U : set of updates) ▷ *read-write transactions*
21: $rs \leftarrow$ GET_ALL(K)
22: write set $ws \leftarrow \emptyset$
23: **for** key $k \in K$ **do** $ws[k] \leftarrow u_k(rs[k]) : u_k \in U$
24: invoke PUT_ALL(ws)

Partition-side Data Structures

25: $versions$: set of versions ⟨key, $value$, timestamp ts, metadata md⟩
26: $latest[k]$: last committed timestamp for key k

Partition-side Methods

27: **procedure** GET(k : key, ts_{ra} : timestamp)
28: $val \leftarrow v.value : v \in versions \wedge v.key = k \wedge v.ts = ts_{ra}$
29: $met \leftarrow w.md : w \in versions \wedge w.key = k \wedge w.ts = latest[k]$
30: **return** ⟨k, val, $latest[k]$, met⟩ ▷ *return latest committed timestamp rather than val's timestamp as in RF*

31: **procedure** PREPARE(v : version)
32: $versions$.add(v)
33: **return**

34: **procedure** COMMIT(ts_c : timestamp)
35: $K_{ts} \leftarrow \{v.key \mid v \in versions \wedge v.ts_v = ts_c\}$
36: **for** $k \in K_{ts}$ **do** $latest[k] \leftarrow \max(latest[k], ts_c)$

session guarantee: when the coordinator computes RA-consistent snapshots for transactional reads, previous writes have already been incorporated in the snapshots.

Unlike RAMP-Fast that considers only read-only and write-only transactions, LORA supports read-write transactions with mixed reads and writes. LORA starts a read-write transaction with the UPDATE method (lines 20-24). It first invokes the GET_ALL method for transactional reads to retrieve the values of the keys the coordinator wants to update (stored then in the read set), as well as the corresponding last seen timestamps and metadata. LORA then updates the write set accordingly and proceeds with the PUT_ALL method for transactional writes.

It is worth mentioning that LORA imposes no more overhead than RAMP-Fast in terms of message payload for both reads (i.e., one version returned per read) and writes (i.e., metadata containing sibling keys). Naively returning far stale versions, e.g., the initial versions, may satisfy RA, but users prefer recent data [33, 79]. LORA improves data freshness by constantly updating the coordinator's view of latest committed versions during the processing of both reads (line 30) and writes (line 16); any subsequent RA-consistent snapshot is therefore able to incorporate such versions. See Section 7.4.5 for our statistical analysis of data freshness in LORA.

By Example. The following example illustrates how LORA works, focusing on its one round-trip, read atomic reads. We use the same scenario as in Example 3.1 to emphasize the difference from RAMP-Fast.

Example 3.2 (Fig. 3-(b)). Assume that both coordinators initially have the initial views of the data items x and y, i.e., $last[x] = \langle 0, \{y\}\rangle$ and $last[y] = \langle 0, \{x\}\rangle$. A write-only transaction $T_1 : [w(x_1),\ w(y_1)]$, issued by coordinator C_1, is attempting writes to both keys, while a read-only transaction $T_2 : [r(x),\ r(y)]$, issued by coordinator C_2, proceeds while T_1 is writing. Analogously, T_2 reads from P_x after P_x has committed T_1's write to x (indicated by $latest[x] = 1$) while from P_y before P_y has committed T_1's write to y ($latest[y] = 0$). With LORA T_2 now requires only one round trip to return the RA-consistent versions x_0 and y_0 which have been specified by timestamp $ts_{ra} = 0$ in both reads to P_x and P_y, respectively. Timestamp $ts_{ra} = 0$ is computed based on C_2's local view of last seen timestamps of x and y (e.g.,). T_2 is compatible with T_1 that now commits after the PREPARE phase.

In the above example, upon committing T_1, coordinator C_1 incorporates the transactional writes into its view of last seen timestamps ($last[x] = \langle 1, \{y\}\rangle$ and $last[y] = \langle 1, \{x\}\rangle$). A subsequent read-only transaction accessing x and y can return both RA-consistent and RYW-consistent versions x_1 and y_1 even though it is racing with T_1's COMMIT phase. See Appendix B for the visualized scenario, as well as the comparison with the conjectured design 1PW that also commits writes in one phase but violates RYW.

4 THE MAUDE ECOSYSTEM

In this section we give preliminaries on rewriting logic, Maude, and statistical model checking of probabilistic rewrite theories that are used in our formalization and verification of LORA (Section 5–7).

4.1 Rewriting Logic and Maude

Maude [22] is a rewriting-logic-based executable formal specification language and high-performance analysis tool for object-based distributed systems. The tool provides a wide range of automatic formal analysis methods, including simulation, reachability analysis, and linear temporal logic (LTL) model checking. Maude has been very successful in analyzing designs of a wide range of distributed and networked systems [16, 46, 48, 82, 83].

A Maude module specifies a *rewrite theory* [56] $(\Sigma, E \cup A, L, R)$, where

- Σ is an algebraic *signature*; i.e., a set of *sorts*, *subsorts*, and *function symbols*;
- $(\Sigma, E \cup A)$ is a *membership equational logic theory* [22], with E a set of possibly conditional equations and membership axioms, and A a set of equational axioms such as associativity, commutativity, and identity, so that equational deduction is performed *modulo* the axioms A. The theory $(\Sigma, E \cup A)$ specifies the system's states;
- L is a set of rule labels;
- R is a collection of *labeled conditional rewrite rules* of the form $[l] : t \longrightarrow t'$ **if** *cond*, with $l \in L$, that specify the system's transitions.

We briefly summarize Maude's syntax and refer to [22] for more details. Sorts and subsort relations are declared by the keywords **`sort`** and **`subsort`**, and operators (or functions) are introduced with the **`op`** keyword: **`op`** f `:` $s_1 \ldots s_n$ `->` s, where $s_1 \ldots s_n$ are the sorts of its arguments, and s is its (value) *sort*. Operators can have user-definable syntax, with underbars '`_`' marking each of the argument positions, and are declared with the sorts of their arguments and the sort of their result. Some operators (e.g., the union operator '`,`' for maps and sets) can have equational *attributes*, such as **`assoc`**, **`comm`**, and **`id`**, stating that the operator is associative and commutative and has a certain identity element (e.g., `empty` for maps and sets). Such attributes are then used by the Maude engine to match terms *modulo* the declared axioms. An operator can also be declared to be a *constructor* (**`ctor`**) that defines the data elements of its sort. Declaring an operator f with the **`frozen`** attribute forbids rewriting with rules in all proper subterms of a term having f as its top operator.

There are three kinds of logical statements in the Maude language, *equations*, *memberships* (declaring that a term has a certain sort), and *rewrite rules*, introduced with the following syntax:

- equations: **`eq`** $u = v$ or **`ceq`** $u = v$ **`if`** *condition*;
- memberships: **`mb`** $u : s$ or **`cmb`** $u : s$ **`if`** *condition*;
- rewrite rules: **`rl`** `[`l`]:` u `=>` v or **`crl`** `[`l`]:` u `=>` v **`if`** *condition*.

An equation $f(t_1, \ldots, t_n) = t$ with the **`owise`** (for "otherwise") attribute can be applied to a term $f(\ldots)$ only if no other equation with left-hand side $f(u_1, \ldots, u_n)$ can be applied. The mathematical variables in such statements are either explicitly declared with the keywords `var` and `vars`, or can be introduced on the fly in a statement without being declared previously, in which case they have the form *var*`:`*sort*. Maude also provides standard parameterized data types (sets, lists, maps, etc.) that can be instantiated (and renamed). For example, `Map{Nat,String}` defines a mapping m from natural numbers to strings, having each entry of the form n `|->` s; the lookup operator m`[`n`]` returns the n-indexed string s. Finally, a comment is preceded by '`***`' and lasts till the end of the line.

In object-oriented Maude specifications, a *class* declaration **`class`** C `|` att_1 `:` s_1`, . . . ,` att_n `:` s_n declares a class C of objects with attributes att_1 to att_n of sorts s_1 to s_n. An *object instance* of class C is represented as a term `<` O `:` C `|` att_1 `:` $val_1, \ldots,$ att_n `:` val_n `>`, where O, of sort `Oid`, is the object's *identifier*, and where val_1 to val_n are the current values of the attributes att_1 to att_n. A *message* is a term of sort `Msg`. A system state is modeled as a term of the sort `Configuration`, and has the structure of a *multiset* made up of objects and messages built up with an empty syntax (juxtaposition) multiset union operator `__`.

The dynamic behavior of a system is axiomatized by specifying each of its transition patterns by a rewrite rule. For example, the rule `[l]`

```
rl [l] :  (to O from O' : w)  < O : C | a1 : x, a2 : y, a3 : z >
     =>
          < O : C | a1 : x + w, a2 : y, a3 : z >  (to O' from O : z - w) .
```

defines a family of transitions in which a message (`to O from O' : w`) sent by the object `O'` is read and consumed by the receiving object `O` of class `C`, whose attribute `a1` is updated to `x + w` with the message payload `w`, and a new message (`to O' from O : x`) is generated. Attributes whose values do not affect the next state, such as `a2`, need not be mentioned in a rule. Attributes whose values do not change, such as `a3`, can be omitted in the right-hand side of a rule. Hence, the above rewrite rule can also be written as:

```
rl [l] :  (to O from O' : w)  < O : C | a1 : x, a3 : z >
      =>
          < O : C | a1 : x + w >  (to O' from O : z - w) .
```

Example 4.1. The following Maude module `QUERY` specifies a very simple distributed database system where each client performs a sequence of queries on the keys stored across different database partitions. Rule `[read]` (lines 14–17) describes when a client (object of class `Client`; line 5) wants to read the value of key `K`, it issues a query `read(K)` (defined in line 8) from its buffered queries to the corresponding database partition (object of class `DB`; line 6) computed by looking up the local key-database mapping `KD[K]`. This database partition replies with the stored key-value pair (defined in line 8) upon receiving the query (rule `[reply]`; lines 18–21). When the client reads the response, it appends the associated key-value pair `< K,V >` to its log (rule `[log]`; lines 22–25). Note that, despite the message payload sort of `Payload` (line 9), both queries and key-value pairs can be piggybacked via messages (e.g., in rule `[reply]`) due to the subsort relations (line 3).

```
1  mod QUERY is
2  *** user-defined sorts and subsorts
3  sorts Query Key Value KeyValuePair Payload .  subsorts Query KeyValuePair < Payload .

4  *** class declarations for client and database
5  class Client | queries : List{Query}, log : List{KeyValuePair}, mapping : Map{Key,Oid} .
6  class DB | database : Map{Key,Value} .

7  *** user-defined constructors
8  op read : Key -> Query [ctor] .              op <_,_> : Key Value -> KeyValuePair [ctor] .
9  op to_from_:_ : Oid Oid Payload -> Msg [ctor] .

10 *** variable declarations
11 vars O O' : Oid .    var K : Key .    var QS : List{Query} .    var KD : Map{Key,Oid} .
12 var V : Value .      var B : Map{Key,Oid} .    var LOG : List{KeyValuePair} .

13 *** rewrite rules for system dynamics
14 rl [read] :
15    < O : Client | queries : read(K) QS, mapping : KD >
16 =>
17    < O : Client | queries : QS >    (to KD[K] from O : read(K)) .

18 rl [reply] :
```

```
19     (to O from O' : read(K))    < O : DB | database : B >
20  =>
21     < O : DB | >    (to O' from O : < K,B[K] >) .

22  rl [log] :
23     (to O from O' : < K,V >)    < O : Client | log : LOG >
24  =>
25     < O : Client | log : LOG < K,V > > .
26  endm
```

The following shows an example system state (of sort `Configuration`) with two clients (`c1` and `c2`), each having two queries, and two database partitions (`db1` and `db2`), each storing two data items:

```
1  *** constant object identifiers: c1, c2, db1, and db2; constant keys: k1, k2, k3, and k4
2  (to c1 from db1 : < k2,8 >)    (to db2 from c2 : read(k4))
3  < c1 : Client | queries : nil, log : < k3,9 >,
4                  mapping : k1 |-> db1, k2 |-> db1, k3 |-> db2, k4 |-> db2 >
5  < c2 : Client | queries : read(k3), log : nil,
6                  mapping : k1 |-> db1, k2 |-> db1, k3 |-> db2, k4 |-> db2 >
7  < db1 : DB | database : k1 |-> 54, k2 |-> 8 >    < db2 : DB | database : k3 |-> 9, k4 |-> 7 >
```

where `c1` has finished its first query, indicated by the key-value pair `< k3,9 >` in the log (line 3), and is ready to read the returned data from `db1` for the second query (the first message in line 2); `c2` has just issued its first query `read(k4)` to `db2` (the second message in line 2) with the next query `read(k3)` buffered (line 5).

4.2 Probabilistic Rewrite Theories and Statistical Model Checking

Probabilistic distributed systems can be modeled as *probabilistic rewrite theories* [1] in Maude with rules of the form

$$\texttt{[l]}: \quad t(\overrightarrow{x}) \longrightarrow t'(\overrightarrow{x},\overrightarrow{y}) \ \textbf{if}\ cond(\overrightarrow{x})\ \textit{with probability}\ \overrightarrow{y} := \pi(\overrightarrow{x})$$

where the term t' has additional new variables $\overrightarrow{y}$ disjoint from the variables $\overrightarrow{x}$ in the term t. For a given matching instance of the variables $\overrightarrow{x}$ there can be many ways to instantiate the extra variables $\overrightarrow{y}$. Due to such potential nondeterminism, probabilistic rewrite rules are *not directly executable*. By sampling the values of $\overrightarrow{y}$ according to the probability distribution $\pi(\overrightarrow{x})$ (that depends on the matching instance of $\overrightarrow{x}$), they can be simulated in Maude (see Section 7.4.1 for a concrete example).

Statistical model checking (SMC) [69, 88] is a formal approach to analyzing probabilistic systems against temporal logic properties. Compared to conventional simulations or emulations, SMC can verify a property specified, e.g., in a stochastic temporal logic (QuaTEx [1] used in our analysis), up to a user-specified *statistical confidence level* by running Monte-Carlo simulations of the system model. The expected value $\bar{v}$ of a property query belongs to the interval $[\bar{v}-\frac{\beta}{2}, \bar{v}+\frac{\beta}{2}]$ with $(1-\alpha)$ statistical confidence, where the parameters α and β (indicating the margin of error) determine when an SMC analysis stops performing simulations [69]. An SMC result can be a probability or percentage with respect to some qualitative property such as "99.1% of LORA reads return the latest values written to the database with

95% confidence", or a quantitative estimation of some performance property such as "With 99% statistical confidence, the average latency of LORA transactions is 1.5 time units".

Maude-based SMC with the PVeStA tool [5] has been successfully used in statistically analyzing high-level designs of distributed database systems. In particular, with significantly less effort than the actual system implementations (e.g., 20x less in terms of lines of code [51]), SMC-based performance estimations and predictions in an early design phase have shown good correspondence with implementation-based evaluations under realistic deployment [16, 48, 51].

5 FORMALIZING LORA

This section presents a formal model of LORA in Maude. The entire executable specification, consisting of 870 lines of code, is available at [43].

5.1 Data Types, Objects, and Messages

We model LORA in the CAT framework [49] for formally specifying distributed transaction systems. This enables us to leverage the CAT model checker to verify LORA's data consistency properties (see Section 7.3).

More specifically, LORA is formalized in an object-oriented style, where the state is a multiset consisting of a number of *partition* objects, each modeling a partition of the entire database, a number of *coordinator* objects, each modeling the proxy of a client that executes *transactions* formalized also as objects residing inside the coordinator, and a number of *messages* traveling between the objects.

Data Types. A *version* is modeled as a 4-tuple `< key, value, timestamp, metadata >` (of sort `Version`) consisting of the *key*, its *value*, and the version's *timestamp* and *metadata*. A timestamp is modeled as a pair `< id, sqn >` consisting of an object identifier and a local sequence number that together uniquely identify a version. Metadata are modeled as a set of keys, indicating, for each key, the sibling keys written in the same transaction. The partition-side data structure *versions* (line 25, Algorithm 1) is modeled as a set of versions, denoted by `Set{Version}`, that instantiates Maude's parameterized container *sets* on sort `Version`. We also use another container *maps* to define *latest* (line 26) as `Map{Key,Timestamp}` mapping *keys* to *timestamps*. The coordinator-side data structure *last* (line 1) can be similarly defined as a mapping from each key to a pair `< timestamp, metadata >` (of sort `tmPair`).

Objects. A partition stores parts of the entire database. We formalize it as an object instance of the following class `Partition`:

```
class Partition | database : Set{Version}, latest : Map{Key,Timestamp} .
```

where the partitioned `database` consists of a set of versions for each key stored locally. The partition also retains the timestamp of the `latest` committed version for each key.

A transaction is modeled as an object instance of the class `Txn`:

```
class Txn | operations : List{Operation}, waitinglist : Set{Oid},
            readset : Set{kvPair}, writeset : Set{kvPair} .
```

where `operations` indicates the transaction's reads/writes, with each of the form `read(`*key*`)` or `write(`*key*`,` *value*`)`. The attribute `readset`, resp. `writeset`, denotes the `< key, value >` pairs fetched, resp. written, by the reads, resp. writes. The database partitions from which the coordinator awaits responses with respect to the transaction are stored in `waitinglist`.

A coordinator, modeled as an object instance of the class `Coord`, is delegated to process transactions:

```
class Coord | sqn : Nat, last : Map{Key,tmPair}, queue : List{Object},
              mapping : Map{Key,Oid}, executing : Object, committed : Set{Object} .
```

The attributes queue, `executing`, and `committed` store the transaction object(s) which are waiting to be executed, currently executing, and committed, respectively. Concurrently executing transactions can be modeled by multiple coordinators with each holding a currently executing transaction. The attribute `last` maps each key to the timestamp-metadata pair of its latest version the coordinator has seen. To model the keyspace partitioning we use a `mapping` to pair keys and database partition identifiers.

Initial State. The following shows an example initial state (of sort `Configuration`) of LORA (with some parts replaced by '...') with two coordinators, `c1` and `c2`, buffering, respectively, two and three transactions. Read-only transaction `t1` has two read operations on, respectively, two keys `k1` and `k2`. Write-only transaction `t2` consists of two write operations, with each writing a value (e.g., `"apple"`) to a key (e.g., `k2`). Transaction `t3` is a read-write transaction. The key space is split by three partitions `p1`, `p2`, and `p3`, e.g., keys `k1` and `k3` are stored at `p1`. Each partition is initialized accordingly; in particular, for each key, the value is the empty string, the timestamp is `null`, and the metadata is an `empty` set:

```
op initialState : -> Configuration .
eq initialState =
     < c1 : Coord |
            queue : (< t1 : Txn | operations : (read(k1) read(k2)), waitinglist : empty,
                                  readset : empty, writeset : empty >
                      < t2 : Txn | operations : (write(k2,"apple") write(k3,"pear")), ... >),
            sqn : 0, last : empty, executing : null, committed : empty,
            mapping : (k1 |-> p1, k2 |-> p2, k3 |-> p1, k4 |-> p3) >
     < c2 : Coord |
            queue : (< t3 : Txn | operations : (read(k3) write(k4,"orange")), ... >
                      < t4 : Txn | ... > < t5 : Txn | ... >), ... >
     < p1 : Partition |
            database : (< k1, "", null, empty >, < k3, "", null, empty >), latest : empty >
     < p2 : Partition | ... > < p3 : Partition | ... > .
```

Messages. A message has the form `to` *receiver* `from` *sender* `:` *mp*. The terms *sender* and *receiver* are object identifiers. The term *mp* is the message payload, having the form:

- `get(`*tid*`,`*key*`,`*ts*`)`, for a GET message from transaction *tid* requesting the version *ts* of *key*;
- `reply(`*tid*`,`*key*`,`*val*`,`*ts*`,`*md*`)`, for the returned version from the partition.;
- `prepare(`*tid*`,`*key*`,`*val*`,`*ts*`,`*md*`)`, for preparing the version written by transaction *tid* at the partition;
- `prepared(`*tid*`)`, for confirming a prepared version;
- `commit(`*tid*`,`*ts*`)`, for marking the versions with timestamp *ts* as committed;
- `committed(`*tid*`)`, for acknowledging a successful *commit* operation of transaction *tid*.

5.2 Formalizing LORA's Dynamics

This section formalizes LORA's dynamic behaviors with respect to its SNOW-optimal read-only transactions using rewrite rules.[5] We show the corresponding 4 (out of 12) rules and refer to Appendix C for the remaining rules regarding LORA's write-only and read-write transactions. For each rule, we also refer to the corresponding line(s) of code in Algorithm 1. The complete definitions of the data types and the functions are given in [43].

Issuing GET Messages (Lines 2–6). A coordinator executes a read-only transaction (ensured by the predicate `readOnly` that returns true if a transaction's operations are all reads) if the transaction object appears in `executing` in the left-hand side of a rule:

```
crl [get-all] :
 < C : Coord | executing : < T : Txn | operations : OPS, waitinglist : empty, readset : empty >,
               last : LAS, mapping : KP >
=>
 < C : Coord | executing : < T : Txn | waitinglist : addR(OPS,KP) > >
 getAll(C,T,OPS,LAS,KP)    if readOnly(OPS) .
```

where the attributes `waitinglist` and `readset` must be explicitly present with `empty`. The coordinator adds to the waiting list the partitions from which it expects to receive the returned versions (by function `addR`). The `getAll` function (line 6) iterates the operations of the transaction and generates a set of `get` messages (of sort `Configuration`) directed at the corresponding partitions (the first equation):

```
  op getAll : Oid Oid Operations Map{Key,tmPair} Map{Key,Oid} -> Configuration .
  eq getAll(C,T,(read(K) OPS),LAS,(K |-> P,KP))
   = (to P from C : get(T,K,max(K,LAS)))   getAll(C,T,OPS,LAS,(K |-> P,KP)) .
  eq getAll(C,T,(write(K,VAL) OPS),LAS,KP) = getAll(C,T,OPS,LAS,KP) .
  eq getAll(C,T,nil,LAS,KP) = null .
```

A write operation contributes no message (the second equation). This function returns if every operation of the transaction has been processed (indicated by `nil` in the third equation). An RA-consistent snapshot is computed by the (overloaded) operator `max` (line 5):

```
  op max : Key Map{Key,tmPair} -> Timestamp .
  op max : Timestamp Timestamp -> Timestamp .
```

For each key `K`, the maximum of its last seen timestamp `TS` and (the maximum of) the associated last seen timestamps for which `K` is a sibling (indicated by the matching `K` in the metadata for `K'` in the second equation) is returned (line 5):

```
  eq max(K,(K |-> < TS,MD >,LAS)) = max(TS,max(K,LAS)) .
  eq max(TS,max(K,(K' |-> < TS',(K,KS) >,LAS))) = max(max(TS,TS'),max(K,LAS)) .
  eq max(TS,max(K,LAS)) = TS [owise] .
```

Note that the third equation with the **`owise`** attribute covers all remaining cases after all such `K'` are processed.

Receiving a GET Message (Lines 27–30). Upon receiving a read, the partition replies with a version whose value is indicated by the requested timestamp `TS` and whose timestamp and metadata are, respectively, the latest committed

[5] We do not include variable declarations, but follow the Maude convention that variables are written in (all) capital letters.

timestamp for the requested key and the associated metadata (see [43] for the definitions of the two "match" functions vMatch and mdMatch):

```
rl [rcv-get] :
   (to P from C : get(T,K,TS))
   < P : Partition | database : VS, latest : LAT >
=>
   < P : Partition | >
   (from P to C : reply(T,K,vMatch(TS,KS),LAT[K],mdMatch(K,LAT,VS))) .
```

Receiving a Response (Lines 6–7). When a coordinator receives a reply message of the returned version, it adds the pair < K,VAL > to the transaction's read set, updates last with the latest committed timestamp TS for key K and the associated metadata MD, and removes partition P from the waiting list:

```
rl [rcv-reply] :
   (to C from P : reply(T,K, VAL, TS, MD))
   < C : Coord | executing : < T : Txn | waitinglist : P ; OS, readset : RS >,
                 last : LAS >
=>
   < C : Coord | executing : < T : Txn | waitinglist : OS, readset : (RS, < K, VAL >) >,
                 last : update(K, TS, MD, LAS) > .
```

Committing a Read-Only Transaction (Line 8). A read-only transaction commits at the coordinator by changing its status from executing to committed, when the transaction has collected all replies from the partitions (indicated by the empty waiting list) and its read set is no longer empty (which was initialized as empty; see rule [get-all]):

```
crl [committed-ro] :
 < C : Coord | executing : < T : Txn | operations : OPS, waitinglist : empty, readset : (R, RS) >,
               committed : TXNS >
=>
 < C : Coord | executing : null,
               committed : TXNS < T : Txn | > >    if readOnly(OPS) .
```

6 CORRECTNESS PROOFS FOR LORA

In this section we formally prove that LORA provides *SNOW-optimal* (Section 6.1) and *read atomic* (Section 6.2) transactions based on the Maude model in Section 5. We also refer to Appendix D for the proof of LORA's *read your writes* session guarantee.

Assumptions. We make the following assumptions about the system model as in the formal reasoning about RAMP, SNOW, and transactional data consistency [10, 15, 37]: (1) a timestamp uniquely identifies a version; (2) without loss of generality, versions of the keys written by a transaction are the same; (3) a version is persistent on the partition once it is prepared or committed; (4) all transactions commit unless otherwise noted; (5) each transaction contains at most one write to each key; (6) reads precede writes in read-write transactions; and (7) the network is asynchronous and reliable, i.e., any message will eventually be delivered to its receiver.

Notations. We use P_k to denote the partition storing key k. The partial version order is indicated by $<$. Given two versions v_k and v'_k for key k and the respective timestamps ts_k and ts'_k with $v_k.ts = ts_k$ and $v'_k.ts = ts'_k$, we write $v_k < v'_k$ or $ts_k < ts'_k$ if v'_k appears later than v_k in the version order. Each key k has an initial version $\perp_k$ with $\perp_k < v_k$ for any version v of k. A snapshot taken by a transaction T is a set of timestamps $\{ts_{k_1}, ..., ts_{k_n}\}$, where $k_1, ..., k_n$ written by T are sibling keys to each other and timestamp ts_{k_i} corresponds to version v_{k_i} with $v_{k_i}.ts = ts_{k_i}$. We denote a system execution e, starting from an initial state, by the labels of rules that apply in order: $l_1, ..., l_n$. We use *prefix*(e, l) to indicate the finite prefix of an execution e ending with the firing of rule l.

6.1 Providing SNOW-Optimal Reads

We first adapt the formal definitions of individual SNOW properties in [37] into our rewriting logic setting. Since LORA is designed for read atomic transactions, thus not satisfying *strict serializability* (S), we only need to consider the N+O+W properties for its SNOW-optimality (as well as its latency-optimality; Definition 2.1 and Definition 2.2): *Non-blocking reads* (N), *One-response per read* (O), and *Write transactions that conflict* (W).

The N property indicates that if a coordinator *coord* sends a read request to a partition *part* during the transaction execution, then *part* must respond to *coord* for this request without waiting for any *external* event such as the arrival of a message. We formally define this property as follows.

Definition 6.1 (N). Suppose in an execution e, following rule `[start-txn]`[6] on the initial state and rule `[get-all]` for read-only transactions (resp. rule `[update]` for read-write transactions), rules `[rcv-get]` and `[rcv-reply]`, corresponding to the GET message `to` *part* `from` *coord* `:` *get*(*key*, *ts*), are fired in order with respect to the partition *part* storing *key*. Then there exists an execution e' such that

(i) the execution fragments *prefix*(e,`[rcv-get]`) and *prefix*(e',`[rcv-get]`) are identical;
(ii) in e' rule `[rcv-reply]` is fired at *part* after rule `[rcv-get]` applies without any firing of rules in between for handling *get*(*key*, *ts*).

Note that in (ii) other rule(s) irrelevant to this GET message (or this particular read request) may apply between the firings of rules `[rcv-get]` and `[rcv-reply]`.

The O property states that each read request during the transaction execution must complete successfully in one round of coordinator-to-partition communication *and* exactly one version of the requested key is sent back by the partition. This property is formally defined as:

Definition 6.2 (O). Suppose in a system execution e, after rule `[get-all]` for read-only transactions (or rule [update] for read-write transactions) applies at some coordinator, there exists exactly a pair of firings of rules `[rcv-get]` and `[rcv-reply]` in e for each associated partition P_k with the returned version of the value v_k.

The W property means that transactional reads can be issued at any point, even in the presence of concurrently executing writes that update the same key(s) being read.

Definition 6.3 (W). Suppose in an execution e, rule `[put-all]` for write-only transactions (resp. rule `[finished-rs]` for read-write transactions) is fired, then the transaction can be successfully committed in e, i.e., appearing in the coordinator's attribute `committed`.

LEMMA 6.4. *LORA's reads are non-blocking (N).*

[6]We refer to Section 5.2 for the rules of read-only transactions and Appendix C for the remaining rules.

Proof. By inspection of the formal model of LORA for the partition's response to an incoming GET message (rule [rcv-get]), where the only rule that can be fired with respect to this read request is [rcv-reply]. □

Lemma 6.5. *LORA provides one response per read (O).*

Proof. By inspection of the rules for handling reads: (i) the coordinator issues read requests in parallel to the corresponding partitions (rule [get-all]) and collects all returned versions (rule [rcv-reply]) sent by the partitions (rule [rcv-get]) before committing the read-only transaction (rule [committed-ro]); and (ii) for each GET message on key K, the partition replies with a single version (computed by function vMatch) indicated by timestamp TS where the rest of the return data contain no more version (i.e., only the latest committed timestamp LAT[K] and the sibling keys computed by function mdMatch). The same reasoning applies to the case of read-write transactions. □

Lemma 6.6. *LORA's reads are compatible with conflicting writes (W).*

Proof. By inspection of the rules for handling writes: write-only (resp. read-write) transactions always complete by applying (the if-then branch of) rule [rcv-prepared] after they are issued out (rule [put-all], resp. rule [update]) and appear in committed in the right-hand side of rule [rcv-prepared]. □

According to Lemma 6.4–6.6, LORA satisfies all three properties N+O+W, and thus is SNOW-optimal.

Theorem 6.7. *LORA provides SNOW-optimal reads.*

6.2 Providing Read Atomic Transactions

We first adapt some definitions in [10] with respect to *read atomicity* into our setting.

Definition 6.8 (Read Atomicity). A distributed transaction algorithm satisfies *read atomicity* if it prevents fractured reads anomalies and also prevents transactions from reading uncommitted, aborted, or intermediate data.

Definition 6.9 (Fractured Reads). A transaction T_j exhibits *fractured reads* if transaction T_i writes version v_x and v'_y (in any order, with x possibly but not necessarily equal to y), T_j reads version v_x and version v''_y, and $v'' < v'$."

Definition 6.10 (Aborted Reads). A transaction T_j exhibits *aborted reads* if there exist an aborted transaction T_i such that T_j reads a version written by T_i.

Note that Definition 6.9 subsumes the definition of *intermediate reads* [10].

The key of LORA's read atomic transactions lies in how fractured-reads-free snapshots are computed.

Lemma 6.11. *Any snapshot returned by the* max *function contains no fractured reads.*

Proof. Suppose there exists a snapshot returned by the max function that exhibits fractured reads, and, without loss of generality, it is $\{ts_x, ts'_y\}$. Then the corresponding RA-consistent versions are written by transaction $T = [w(ts_x), w(ts_y), \ldots]$, with "..." the remaining operations on keys rather than x and y, and $ts'_y < ts_y$, thus $ts'_y < ts_x$. By abusing the notation, $y \in ts_x.md$ (and vice versa). According to the definition of max (Section 5.2), ts_x = max(last[x].ts, max({last[k].ts : $x \in$ last[k].md})), and ts'_y = max(last[y].ts, max({last[k].ts : $y \in$ last[k].md})).

Depending on the value of ts_x, we consider the following three cases:

Case 1. If ts_x = last[x].ts, then we have $y \in$ last[x].md, and thus $ts'_y \geq ts_x$, a contradiction.

Case 2. If ts_x = last[y].ts, then last[y] = < ts_x, $md \cup \{x\}$ >, and thus $ts'_y \geq$ last[y].ts, a contradiction.

Case 3. If ts_x = `last[`k`].ts` and $k \notin \{x, y\}$ (i.e., T also includes $w(ts_k)$ where both x and y are k's sibling keys), then we have `last[`k`]` = $< ts_k,\ md \cup \{x, y\} >$, thus $ts'_y \geq$ `last[`k`].ts`, a contradiction. □

Lemma 6.12. *When a read arrives, the requested timestamped version is present on the partition.*

Proof. The timestamp ts_x of a requested version for key x is determined by the `max` function, where ts_x = `last[`x`].ts`, or ts_x = `last[`k`].ts` with $x \in$ `last[`k`].md`. In the case where the requested version is written by another transaction:

Case 1. ts_x = `last[`x`].ts` implies that ts_x has been piggybacked by the coordinator's last read on the key x (rule `[rcv-reply]`), which in turn implies that ts_x was the latest committed timestamp for x (rule `[rcv-get]`). Hence, the version with ts_x had been committed on P_x, and stored in its local `database`.

Case 2. If $x \in$ `last[`k`].md`, then x's sibling key k has already been committed on P_k, and `last[`k`].ts` has been fetched back by the coordinator's last read on k (rule `[rcv-reply]`). This implies that k's sibling version on x with the same timestamp `last[`k`].ts` has been prepared on P_x, and thus stored in its local `database` (by following the rules `[rcv-prepare]` and `[rcv-prepared]`).

In the case where the requested version is written by the read's previous write, the above reasoning applies if there exists another read (in the same session) on x in between these two operations which piggybacks ts_x. Otherwise, if the read follows immediately after the write of timestamp ts_x, then, whether ts_x = `last[`x`].ts` or ts_x = `last[`k`].ts` with $x \neq k$, the corresponding version has been prepared on P_x (rule `[rcv-prepare]`) before the write can be committed (the `if-then` branch in rule `[rcv-prepared]`). By assumption (3), the prepared version is present on P_x. □

Lemma 6.13. *Any snapshot returned by the* `max` *function contains no aborted version.*

Proof. For any timestamp ts returned by the `max` function, it is mapped to some key k, i.e., `last[`k`]` = ts. Hence, it is equivalent to prove that any such timestamp ts corresponds to a committed version. By examining all rewrite rules, `last` can only be updated via two rules `[rcv-reply]` and `[rcv-prepared]`.

Case 1. If timestamp ts is updated by rule `[rcv-reply]`, it means that ts has been piggybacked via the `reply` message from the partition that returns ts as the latest committed timestamp (rule `[rcv-get]`). Hence, the associated version has been marked as committed on the partition (rule `[rcv-commit]`).

Case 2. If timestamp ts for the read is updated by rule `[rcv-prepared]`, the associated version has already been written by a *committed* transaction in the same session (the `if-then` branch in rule `[rcv-prepared]`). □

Lemma 6.14. *All requested versions by a read-only or read-write transaction are stored in its read set when the transaction is committed.*

Proof. The proof is straightforward by following Lemma 6.12, and by applying rules `[get-all]` (resp. `[update]`) and `[rcv-reply]` for read-only (resp. read-write) transactions: a read-only or read-write transaction has the returned key-value pairs stored in its local read set `readset`. □

Theorem 6.15. *LORA provides read atomic transactions.*

Proof. Let us first introduce some notation. Call the state of a concrete coordinator object *active* if it is currently executing a transaction, i.e., the `executing` attribute is not `null`. Let *aobjs* denote a set of active (ground) objects. Let *robjs* denote a set of the remaining (ground) objects, including the database partitions and the rest of the coordinators. Let *msgs* denote a set of (ground) messages. Starting from an initial state (e.g., the example given in Section 5.1), all

reachable states by a sequence of rewrite steps are of the form: *aobjs robjs msgs*, where either *aobjs* or *msgs* (or both) could be empty.

The proof is by induction on the number n of rewrite steps starting from an initial state `initialState` that: for $n = 0$, `initialState` (i) has the above form and (ii) satisfies RA; assuming that (i) and (ii) hold for n, then (i) and (ii) also hold for a state obtained by $n + 1$ rewrite steps from `initialState`. In particular, to prove (ii) we shall examine whether the read set of every committed transaction is RA-consistent.

The base case $n = 0$ is straightforward: `initialState` is a state of the form *robjs*, thus satisfying (i), and each coordinator object in *robjs* has no committed transaction (indicated by `empty` for the `committed` attribute), thus satisfying (ii).

Regarding the induction case, we further consider two cases, depending on whether *aobjs* is empty or not; for each subcase, we examine all possible rules that can be applied to obtain $(n + 1)$th state:

Case 1 (*aobjs* is empty). We consider two cases depending on which rule can apply next. For both cases, there cannot be any newly committed transaction in $(n + 1)$th state:
Case 1-1 (rule `[start-txn]`). In this case some coordinator just starts a transaction which then resides in `executing` after applying this rule. Thus, $(n + 1)$th state has the form *aobj robjs* with a single active object.
Case 1-2 (rule `[rcv-commit]`). By applying this rule, only the partition's state is changed (i.e., `latest` is updated), thus no new committed transaction. $(n + 1)$th state has the form *robjs msgs* with a new `committed` message in *msgs*.

Case 2 (*aobjs* is nonempty). This could happen in any state where at least one coordinator is executing a transaction. We further consider two cases, depending on whether $(n + 1)$th applied rule leads to a committed transaction:
Case 2-1 (no newly committed transaction by applying one of the 12 rules except `[committed-ro]` and `[rcv-prepared]`). Since there is no newly committed transaction by applying $(n + 1)$th rule, the resulting $(n + 1)$th state still has the form: *aobjs robjs msgs*, and each coordinator's attribute `committed` remains the same. Hence, (i) and (ii) hold.
Case 2-2 (a newly committed transaction by applying rule `[committed-ro]` or `[rcv-prepared]`). The proof proceeds with different transaction types:
Case 2-2-1 (write-only transaction with rule `[rcv-prepared]`). This case is trivial as the read set of a write-only transaction is always empty.
Case 2-2-2 (read-only transaction with rule `[committed-ro]`). By Lemma 6.11–6.14, the read set of a committed read-only transaction reflects a RA-consistent snapshot that contains only committed versions.
Case 2-2-3 (read-write transaction with rule `[rcv-prepared]`). Analogously, by those four lemmas, the read set of a committed read-write transaction consists of RA-consistent, committed key-value pairs.

In each of these three cases $(n + 1)$th state is of the form: *aobjs robjs msgs*, where *aobjs* is empty if there is only one active coordinator in nth state. □

SNOW-Optimality and Read Atomicity. Following Theorem 6.15 and Theorem 6.7, we prove that LORA provides SNOW-optimal read atomic transactions, thus latency-optimal read atomic transactions.

Theorem 6.16. *LORA provides SNOW-optimal read atomic transactions.*

Corollary 6.17. *LORA provides latency-optimal read atomic transactions.*

7 QUALITATIVE AND QUANTITATIVE ANALYSIS OF LORA

In this section we present our automated qualitative and quantitative analyses of LORA. We perform *within the same formal framework* three kinds of analyses (also shown in Fig. 1), i.e., standard model checking (Section 7.3), statistical model checking (Section 7.4), and implementation-based testing and evaluation (Section 7.5), with respect to both data consistency properties such as *read atomicity* and performance properties such as transaction latency.

Our goal is twofold: (i) to validate our mathematical proofs via independent machine-checked verification and via implementation-based testing under realistic deployments; and (ii) to explore quantitative aspects of LORA with varying workload via statistical estimation and via actual performance evaluation, which go beyond the qualitative analyses.

7.1 Algorithms for Comparison

As a baseline, we implement in Maude a SNOW-optimal algorithm, called Committed Reads (CR), that satisfies the slightly weaker consistency property *read committed* (RC). For a fair comparison, CR adopts the same data structures as in LORA, e.g., *versions* and *latest*[k], and only differs in system dynamics, e.g., how to process a transactional read and which version to be returned. CR is expected to deliver the best-case performance against any transaction algorithm (e.g., LORA) that provides data consistency stronger than RC (e.g., RA).

Regarding algorithms that offer *read atomicity* (RA), we consider the "fastest" algorithm RAMP-Fast (or RF) on which LORA is built. We also compare LORA with two optimized RAMP designs, One-Phase Writes (or 1PW) and Faster Commit (or FC), that were only *conjectured* in [10]. Despite the potential suboptimality (Section 3.1), both 1PW and FC are expected to outperform RF, thus serving as strong competing *read atomic* algorithms.

Since the above read atomic algorithms do not consider data replication, our comparison excludes RAMP-TAO [21] that layers RAMP transactions atop Facebook's replicated data store TAO. We do not consider distributed transaction algorithms that provide stronger data consistency such as *update atomicity* and *serializability*. These algorithms target completely different consistency guarantees beyond RA and have been empirically shown to underperform RF [10, 48].

7.2 Generating Transaction Workloads

For all three kinds of analyses we implement a *parametric* workload generator to produce *realistic* database workloads: one initial state for each model checking analysis; different initial states for the simulations in an SMC analysis and for the system runs on a cluster. The implementation, consisting of 330 lines of Maude code, is available at [43].

The generator's parameters characterize the well-known Yahoo! Cloud Serving Benchmark (YCSB) [23] and its extension YCSB+T [28], which are the open standards for performance evaluation of database systems. Table 1 lists all these parameters, the numbers of coordinators (`#coords`), partitions (`#partitions`), read-only transactions (`#rtxns`), write-only transactions (`#wtxns`), read-write transactions (`#rwtxns`), operations per transactions (`#ops/txn`), and keys (`#keys`), the percentage of reads in a workload (`%reads`), and the key-access distribution (`distr`),[7] together with their default values and ranges used in our experiments.

The key idea of the implementation is to *probabilistically* generate an initial state each time the `gen`(*parameters*) function rewrites where *parameters* refers to a combination instance of the above parameters. For example, if `%reads=50`, `distr=hotspot` (80% operations accessing 20% keys [23]), and `#keys=100`, the `gen` function will first *uniformly* decide whether an operation is a read or write, and then sample a value v from the *Bernoulli distribution* with $p = 0.8$. If $v \leq 0.8$ (resp. $v > 0.8$), the `gen` function further picks a key *uniformly* from the range $[1, 20]$ (resp. $[21, 100]$).

[7] Key-access distribution is the probability that a read/write operation accesses a certain key.

Table 1. Workload generator's parameters, the default values, and the ranges for different kinds of analyses. "u", "h", and "z" refer to uniform, hotspot (80% operations accessing 20% keys), and zipfian, respectively.

Parameters	Default Value & Range		
	MC (Sec. 7.3)	SMC (Sec. 7.4)	Evaluation (Sec. 7.5)
#coords	2, [2, 4]	25, [5, 50]	25, [5, 50]
#partitions	2, [2, 4]	5, [2, 10]	5, [2, 10]
#rtxns	2, [2, 4]	250, [50, 475]	5000, [1000, 9500]
#wtxns	2, [2, 4]	250, [25, 450]	5000, [500, 9000]
#rwtxns	0, [0, 4]	0, [25, 450]	0, [500, 9000]
#ops/txn	2, [1, 2]	4, [2, 32]	4, [2, 32]
#keys	4, [2, 8]	50, [10, 100]	500, [100, 1000]
%reads	50, [0, 100]	50, [10, 95]	50, [10, 95]
distr	u, {u, h, z}	u, {u, h, z}	u, {u, h, z}

7.3 Model Checking Consistency Properties

This section investigates whether LORA, as well as the competing algorithms, satisfies the expected data consistency properties using standard model checking.

7.3.1 The CAT Model Checker. CAT [49] is a Maude-based tool for model checking data consistency properties of distributed transaction systems like LORA. It currently supports nine transactional consistency properties including RA, but *without* any session guarantee such as RYW.

The consistency properties are formalized in CAT as functions on the "history log" of a completed system run, where a monitor object is automatically added and updated by the tool to record such a log. Specifically, an execution log (of sort Log) maps each transaction (identifier) to a record < *proxy*, *issueTime*, *finishTime*, *committed*, *reads*, *writes* > (of sort Record), with *proxy* its executing coordinator in our case, *issueTime* the starting time at its proxy, *finishTime* the commit/abort times (of sort VectorTime) at different places including the proxy, *committed* a flag indicating whether the transaction is committed, *reads* the key-value pairs read by the transaction, and *writes* the key-value pairs written.

```
*** predefined in CAT
op <_,_,_,_,_,_> : Oid Time VectorTime Bool Set{kvPair} Set{kvPair} -> Record .
pr MAP{Oid,Record} * (sort Map{Oid,Record} to Log) .  *** sort renamed to Log
```

7.3.2 Formalizing Read Your Writes. RYW [77] guarantees that the effect of a write performed in a session is visible to the subsequent reads in the same session. We define RYW in CAT's framework as follows:[8]

```
*** define RYW
op ryw : Log -> Bool .
*** bad situation (i)
ceq ryw(TID1 |-> <C,T,VT,true,(<X,V>,RS),WS>,
        TID2 |-> <C',T',VT',true,RS',(<X,V>,WS')>, LOG) = false if T' < T .
*** bad situation (ii)
ceq ryw(TID1 |-> <C,T,VT,true,(<X,"">,RS),WS>,
        TID2 |-> <C,T',VT',true,RS',(<X,V>,WS')>, LOG) = false if T' < T /\ V =/= "" .
```

[8]The log-based definition of RYW also corresponds to its formal definition in Appendix D.

```
*** remaining cases that satisfy RYW
eq ryw(LOG) = true [owise] .
```

The function `ryw` analyzes RYW by checking whether there was a "bad situation" in which (i) a committed transaction `TID1` has fetched a version written by a different transaction `TID2` that was issued before it (`T' < T`), *or* (ii) the initial value (i.e., the empty string `""`) while it issued a write transaction (on the same key `X`) previously in the same session (by matching the client identifier `C`). Note that the first conditional equation applies to two cases depending on whether the two transactions were issued from the same client (i.e., `C` equals to `C'` or not); neither satisfies RYW. The third equation with the **`owise`** attribute covers all remaining cases that satisfy RYW, including that a committed transactional read returns the value written by its previous write or by a later write issued in a different session.

Table 2. Model checking results for consistency properties. "✓", resp. "×", refers to satisfying, resp. violating, the property.

System Model	Consistency Property		
	RC	RA	RYW
Committed Reads (CR)	✓	×	×
LORA	✓	✓	✓
One-Phase Writes (1PW)	✓	✓	×
RAMP-Fast (RF)	✓	✓	✓
Faster Commit (FC)	✓	✓	✓

7.3.3 Model Checking Analysis. We have extended CAT by adding the above formal definition of RYW to its consistency property library and applied the tool to all five algorithms against RA, RC, and RYW.

All model checking results shown in Table 2 are as expected. In particular, LORA, as well as RF and FC, satisfies all three consistency properties with up to 8 transactions in total, 2 operations per transaction, 4 clients, 4 partitions, and 8 keys; 1PW violates RYW; CR only satisfies RC. This provides an additional independent *validation* of our proofs for LORA's data consistency guarantees via machine-checked analysis. Each model checking analysis took about 1.5 hour (worst case) to execute on a 3.3 GHz Dual-Core Intel Core i7 CPU with 16GB memory.

7.4 Statistical Analysis of LORA

Our mathematical proofs (Section 6), as well as our automated model checking analysis (Section 7.3), establish the desired data consistency guarantees, but offer no quantitative assessment of LORA with respect to, e.g., the system performance and the dynamics of satisfying stronger consistency properties. We therefore investigate the following questions using statistical analysis:

Q1 Are the statistical verification results consistent with our qualitative analyses?
Q2 How well does LORA achieve the strongest data consistency and freshness?
Q3 Can LORA deliver SNOW-optimal performance?
Q4 Can our methodology help avoid detours in developing mature distributed transaction systems?

We answer Q1 by statistically measuring the probabilities of satisfying RA and RYW for LORA. Regarding Q2, inspired by the work on benchmarking data freshness in distribute key-value stores (with no support for *transactions*) [7, 33, 44, 81], we investigate the probability of LORA's reads returning the latest writes, the most aggressive data freshness [79]. Answers to such questions can guide decisions on the adoption of a given cloud database system: knowing the likelihood that the system will achieve certain stronger data consistency is crucial for some web applications and

could affect user engagement and commercial revenue [33, 81]. To answer Q3 we compare LORA with the competing read atomic algorithms in terms of transaction latency and throughput. For Q4, we showcase an alternative read atomic design (our first attempt in the development course) that also completes the transactional reads in one round trip.

7.4.1 Generating Probabilistic Model. The model in Section 5 are untimed and nondeterministic, which is not well-suited for quantitative analysis with SMC. Hence, we transform it into a timed, purely probabilistic model by following the systematic methodology in [1]. In particular, the transformation assigns to each message a delay sampled from a continuous probability distribution, which determines the firing of the rule receiving the message. The resulting model is therefore free from unquantified nondeterminism and can be simulated by the original model (i.e., faithful behavioral correspondence). In our SMC experiments we use the *lognormal* distribution (with parameters $\mu = 0.0$ and $\sigma = 1.0$) that characterizes the network latency in cloud data centers [11].

We exemplify the transformation with rule `[rcv-get]`. In the following transformed probabilistic rule (with the additions to the original rule written in purple bold), upon receiving the `get` message at global time `GT`, the partition sends back the corresponding version with message delay `D` sampled from the lognormal distribution:

```
rl [rcv-get-prob] :
   {GT, (to P from C : get(K,TS))}
   < P : Partition | database : VS, latest : LAT >
 =>
   < P : Partition | >
   [D, (from P to C : reply(K,vMatch(TS,VS),LAT[K],mdMatch(K,LAT,VS)))]
   with delay D := lognormal(μ,σ,K,TS,VS,LAT) .
```

Since the message delay also accounts for the message payload size, the distribution is additionally parametric on the corresponding variables (i.e., `K`, `TS`, `VS`, and `LAT`) in the left-hand side of the rule which are included in payload `reply`. The outgoing message must then be consumed by receiver `C` when time `D` elapses, i.e., at global time `GT + D`. The elapse of message delays, as well as advancing the global clock, is maintained by a *scheduler* object (see [43] for details).

7.4.2 Extracting Transaction Measures. For our statistical analysis of various properties, we add to the state a *monitor* (similar to the monitor object defined in CAT [49]; also see Section 7.3) which stores in its log crucial information about each transaction, i.e., its identifier, issue time, commit time, and values read/written. Since LORA is terminating if a finite number of transactions are issued, we check the consistency and performance properties by inspecting this monitor object in the final states, when all transactions have finished.

The monitoring mechanism modifies a probabilistic model by adding and updating the monitor object, as well as the scheduler object, whenever needed. We exemplify the modification with rule `[committed-ro]` where the additions are written in purple bold:

```
crl [committed-ro-monitor] :
  < M : Monitor | log : T |-> < C, GT', GT'', FLAG, RS', WS >, LOG >
  < S : Scheduler | clock : GT >
  < C : Coord | executing : < T : Txn | operations : OPS, waitinglist : empty, readset : (R, RS) >,
                committed : TXNS >
=>
  < M : Monitor | log : T |-> < C, GT', GT, true, (R, RS), WS >, LOG >
```

```
< S : Scheduler | >
< C : Coord | executing : null, committed : TXNS < T : Txn | > >    if readOnly(OPS) .
```

When transaction T has finished all the reads, the monitor M records the fetched values (R, RS) stored in readset and the commit time GT extracted from the scheduler object S, and sets the "committed" flag to true. Note here that *finishTime* only denotes when the transaction is finished at the coordinator, instead of the times at different sites as in CAT (see VectorTime in Section 7.3.1), since LORA, as well as the competing algorithms, assumes no data replication, thus no commits at multiple database partitions.

7.4.3 Defining Performance and Consistency Properties. With the extracted transaction measures in the log, we can define a number of functions on states with the monitor for both performance and consistency properties. The following shows the formal definitions of *average latency*, *throughput*, and *latest freshness*. We refer to [43] for the adapted definitions of RA and RC from the CAT framework [49] and of RYW from Section 7.3.2.

Average Latency. The function latency computes the average transaction latency by dividing the sum of the latencies of all committed transactions by the total number of such transactions:

```
op latency : Configuration -> Float [frozen] .
eq latency(< M : Monitor | log : LOG > CF) = totalLatency(LOG) / totalNumber(LOG) .
```

where the term CF denotes the rest of the entire configuration (e.g., the coordinator and partition objects). The (overloaded) function totalLatency iterates the transactions in the log and computes the sum of all their latencies, each indicating the time between the issue time and finish time (e.g., T2 - T1 in the second equation) of a *committed* transaction (indicated by true):

```
op totalLatency : Log -> Float .    op totalLatency : Log Float -> Float .
*** auxiliary function initializing the total latency
eq totalLatency(LOG) = totalLatency(LOG,0.0) .
*** add the latency of a committed transaction
eq totalLatency((TID |-> <C, T1, T2, true, RS, WS>, LOG), L) = totalLatency(LOG, T2 - T1 + L) .
*** exclude any aborted transaction
eq totalLatency((TID |-> <C, T1, T2, false, RS, WS>, LOG), L) = totalLatency(LOG, L) .
*** all transactions have been processed
eq totalLatency(empty, L) = L .
```

Function totalLatency returns after all the transactions are processed (indicated by empty in the fourth equation). The totalNumber function can be defined similarly to compute the size of the log with only the true records, i.e., the number of committed transactions.

Throughput. The throughput function computes the number of committed transactions per time unit:

```
op throughput : Configuration -> Float [frozen] .
eq throughput(< M : Monitor | log : LOG > CF) = totalNumber(LOG) / totalTime(LOG) .
```

where the totalTime function returns the time when all transactions, including the aborted ones, are finished, i.e., the largest *finishTime* in the log:

```
op totalTime : Log -> Float .    op totalTime : Log Float -> Float .
eq totalTime(LOG) = totalTime(LOG,0.0) .
```

```
*** the maximum of TID's finish time and the current largest finish time
eq totalTime((TID |-> <C, T1, T2, FLAG, RS, WS>, LOG), L) = totalTime(LOG, max(T2, L)) .
eq totalTime(empty, L) = L .
```

Latest Freshness. The most aggressive data freshness is *latest freshness* [79], meaning that reads always return the latest committed writes. We define function `latest` to compute the fraction for such read transactions:

```
op latest : Configuration -> Float [frozen] .
eq latest(< M : Monitor | log : LOG > CF) = latReads(LOG) / totalReads(LOG) .
```

Function `latReads` iterates all the committed read-only and read-write transactions (extracted from the log by function `r` in the first equation) and returns the number of transactions whose reads fetch the latest writes:

```
op latReads : LOG -> Float .    op latReads : LOG LOG Float -> Float .
*** extract read-only and read-write transactions
eq latReads(LOG) = latReads(r(LOG),LOG,0.0) .
*** return stale write
ceq latReads((TID1 |-> < C, T1, T2, true, (< X, V >, RS), WS >, LOG),
             (TID2 |-> < C', T3, T4, true, RS', (< X, V >, WS') >,
              TID3 |-> < C'', T5, T6, true, RS'', (< X, V' >, WS'') >, LOG'), N)
  = latReads(LOG, (TID2 |-> < C', T3, T4, true, RS', (< X, V >, WS') >,
                   TID3 |-> < C'', T5, T6, true, RS'', (< X, V' >, WS'') >, LOG'), N)
    if T5 > T3 /\ T5 < T1 /\ V =/= V' .
*** return stale initial value
ceq latReads((TID1 |-> < C, T1, T2, true, (< X, "" >, RS), WS >, LOG),
             (TID2 |-> < C', T3, T4, true, RS', (< X, V >, WS') >, LOG'), N)
  = latReads(LOG, (TID2 |-> < C', T3, T4, true, RS', (< X, V >, WS') >, LOG'), N)
    if T1 > T3 .
*** return the latest write
eq latReads((TID1 |-> < C, T, VT, true, RS, WS >, LOG), LOG', N)
 = latReads(LOG, LOG', N + 1.0) [owise] .
eq latReads(empty, LOG', N) = N .
```

The second (conditional) equation excludes a transaction `TID1` that reads a stale version `V` written by transaction `TID2` (indicated by the matching key-value pair `< X, V >` appearing in `TID1`'s read set and `TID2`'s write set); the latest version `V'` was written by `TID3` that was issued after `TID2` (indicated by `T5 > T3`). In the case where `TID1` reads the initial value, it violates *latest freshness* if there exists a transaction `TID2` that has written to the same key (the third equation). Except such "bad situations" any read must return the latest write (the fourth equation).

7.4.4 Experiment Setup. We employed 50 d430 Emulab machines [85], each with two 2.4 GHz 64-bit 8-Core E5- 2630 processors, to parallelize SMC with PVeStA [5]. We set the statistical confidence level to 95%, i.e., $\alpha = 0.05$, and the error margin β to 0.01 for all our experiments.

Table 3. Probability of achieving *latest freshness* under varying read/write proportion for three representative key-access distributions (e.g., *hotspot* indicates 80% operations accessing 20% keys; *Zipfian* represents a distribution of probabilities of key access that follows Zipf's law) in YCSB [23] and YCSB+T [28]. High probabilities are colored: 90%–95% in yellow and >95% in pink.

Key-Access Distribution	Write-Heavy (10% Reads) CR | LO | 1PW | RF | FC	Medium (50% Reads) CR | LO | 1PW | RF | FC	Read-Heavy (95% Reads) CR | LO | 1PW | RF | FC
zipfian	22.5% | 25.1% | 20.2% | 10.1% | 10.2%	25.1% | 26.3% | 28.1% | 27.7% | 27.7%	64.6% | 29.7% | 56.8% | 53.6% | 57.1%
hotspot	91.7% | 99.9% | 93.1% | 88.3% | 87.5%	93.3% | 90.1% | 93.3% | 94.1% | 94.1%	99.5% | 86.7% | 99.8% | 99.7% | 99.6%
uniform	96.7% | 99.9% | 96.7% | 96.7% | 97.5%	98.7% | 98.8% | 98.7% | 98.7% | 99.1%	99.9% | 97.1% | 99.9% | 99.9% | 99.9%

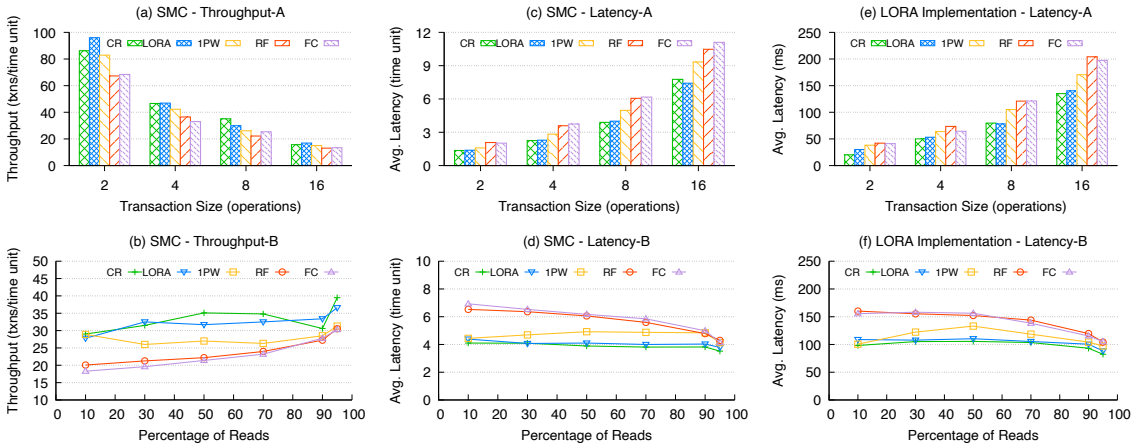

Fig. 4. Performance comparison in terms of transaction throughput and latency with varying transaction size and ratio of reads. Plots (a-d) depict SMC-based estimations while plots (e-f) implementation-based evaluations under realistic deployments. Time units are defined as logical clock ticks in our probabilistic model.

7.4.5 SMC Results. We have subjected to PVeStA the transformed probabilistic models with monitoring, the performance and consistency metrics, and the initial states produced by the workload generator, and performed four sets of experiments according to our research questions. Each SMC analysis took up to 30 minutes (worst case) to terminate.

Experiment 1: Statistically Verifying RA and RYW. In all our SMC analyses, the probabilities of satisfying RA and RYW are 100% for LORA. This provides a strong independent *validation* of our proofs for the data consistency guarantees and of the *model transformation* (from the nondeterministic, untimed model into the timed, purely probabilistic model) via machine-checked analysis.

Experiment 2: Probability of Achieving Latest Freshness. Table 3 shows the probabilities of achieving *latest freshness* under varying read/write proportion for three representative key-access distributions in YCSB [23] and YCSB+T [28]. Despite the weak consistency guarantees, LORA, as well as the competing algorithms, can actually achieve much higher data freshness if the key access is not much skewed (as in *Zipfian*). In particular, if keys are touched *uniformly*, the reads can return almost the latest writes. Unlike the other algorithms, LORA delivers fresher data with more writes, while most of its reads (86.7%) can still fetch the latest writes with the *hotspot* distribution under read-heavy workloads.

Experiment 3: Performance Comparison. Our SMC-based estimations show that LORA outperforms all competing *read atomic* transaction algorithms, i.e., RF, 1PW, and FC, and almost achieves the best-case performance delivered by CR. Fig. 4 (c-d) depict the average latency as a function of transaction size (i.e., `#ops/txn`) and percentage of reads (i.e., `%reads`), respectively. The plots show that our LORA design incurs less latency than the competing algorithms in *all* cases. Fig. 4 (a-b) demonstrate that LORA also provides higher system throughput than the competitors under *all* workloads that we experiment with. For both transaction latency and throughput, LORA is competitive with CR.

Experiment 4: Comparing LORA with the Alternative Design. In our development course we have attempted an alternative design (abbr. as ALT) that also finishes *read atomic* reads in one round trip. Compared to LORA, ALT also piggybacks the timestamp and metadata of the fetched version, in addition to the last seen timestamp and the associated metadata. Even though ALT satisfies the expected data consistency properties (also checked by our extended CAT tool), the SMC-based performance predictions (see Fig. 7 in Appendix E) show that it underperforms LORA with varying workload. Hence, we decided not to proceed with ALT's implementation, deployment, and evaluation.

7.5 Implementation, Deployment, Testing, and Evaluation of LORA

In this section we (i) examine the actual LORA runs under realistic deployments for any RA or RYW violation, (ii) compare the deployed LORA and competing algorithms with respect to the system performance, and (iii) investigate whether the evaluations are consistent with the model-based statistical predictions from Section 7.4.

Regarding (iii), the desired consistency between SMC-based predictions and implementation-based evaluations is an agreement between predicted and measured trends [44, 51]: If, e.g., latency decreases as a function of the proportion of reads, then consistency means that it should do so along curves that are similar up to a change of scale.

7.5.1 Automatic Implementation and Deployment. We use the *D* transformation [51] to generate a *correct-by-construction* distributed Maude implementation from the formal model of LORA (Section 5). Compared to an implementation in a different programming language such as Java, the *D*-transformed implementation preserves the correctness properties (i.e., any *CTL** temporal logic property free from the "next" operator) satisfied by the original model [51, Theorem 3]. In our case, this means that all consistency properties, including RA, RC, and RYW, model checked in Section 7.3 are preserved in our distributed LORA implementation.

Moreover, we apply the same *monitoring mechanism* in Section 7.4 to the transformed LORA implementation. In particular, the monitor object records the relevant transaction information during each distributed system run. We then aggregate and inspect the logs across different machines to compute the overall consistency and performance metrics of the LORA transaction system.

Finally, we optimize the deployment tool in [51] to fully automate the process of deploying and running the distributed LORA implementation in the cloud (a cluster of 10 d430 Emulab machines [85]). More specifically, our new deploying tool takes as input the IP addresses of the distributed machines and the number of Maude sessions on each machine, and initializes each server (i.e., database partition), each client session, and the TCP/IP connections between them from a single cluster machine.

7.5.2 Evaluation and Testing Results. Fig. 4 (e), resp. (f), plots the side-by-side comparison with our SMC-based estimations in Fig. 4 (c), resp. (d). Both plots in each comparison agree reasonably well: (i) LORA outperforms the other *read atomic* algorithms and can compete with CR in all cases; (ii) with the increasing transaction size all algorithms incur higher latency; and (iii) with more reads the average transaction latency tends to decrease.

Additionally, in all collected logs (10^5 transaction records, 250 logs in total) we have not found any RA, RC, or RYW violation for LORA. These testing results further provide a strong independent validation of our qualitative analyses and of the correctness of the generated implementation.

8 DISCUSSION

We have demonstrated that, with our methodology and toolkit, a provably correct distributed transaction algorithm with predicted high performance can be passed from its formal design to the implementation and deployment with empirically confirmed consistency guarantee and performance.

Limitations and Improvements. There is still room for improvement to our toolkit. The following discusses some potential optimizations to each part of the toolkit:

- *Mechanizing proofs.* We can mechanize the pencil-and-paper proofs using the constructor-based reachability logic theorem prover for rewrite theories [73]. Other proof assistants such as Coq [14] are also possible options, which would, however, introduce a semantic gap.
- *Extending CAT.* We can further extend CAT's consistency property library by considering not only other transactional consistency guarantees such as *transactional causality* but also non-transactional consistency models such as *eventual consistency* for NoSQL key-value stores like Cassandra [19]. These consistency properties can be specified analogously as functions on the history log as for RYW (Section 7.3.2).
- *Automating SMC analysis.* There are currently two manual efforts in our SMC analysis: (i) the transformation from a nondeterministic untimed system model (e.g., the Maude model of LORA in Section 5) to a probabilistic timed model (e.g., the transformed model in Section 7.4.1) is performed by hand; and (ii) the monitoring mechanism is manually added to the probabilistic model. To automate the entire process for SMC analysis we can implement a Maude meta-level [22] tool for both transformations and still remain in the same semantics framework.
- *Optimizing Maude Implementations.* The D-transformed Maude implementations may be suboptimal compared to those in sophisticated programming languages such as C++ (also observed in [51]).[9] Optimizing the socket library and the data marshaling/unmarshaling in the Maude language could help resolve the performance issue.

Generality. Our methodology is fairly generic and can be applied to the "full-stack" development of distributed systems in general with certain extensions. Take network routing protocols for example, the CAT's property library could be extended with the *loop freedom* property; the workload generator could be modified accordingly to probabilistically produce route requests; the monitoring mechanism could be generalized to automatically record timed events such as when a source node broadcasts a route request.

Flexibility. Our toolkit can also be customized to different development needs. In the case of LORA, this means that the developers can alternatively implement the algorithm in a conventional programming language (e.g., for the integration into some production database like Facebook's TAO) with a high confidence in both correctness and performance, already obtained from the design-level formal analyses; despite the currently unoptimized prototype, the developers can still use it to quickly explore the actual performance of the algorithm in realistic networking environments, thanks to the automation of code generation and deployment.

[9] Existing formal approaches that extract correct implementations from the formal models (e.g., the Coq-based frameworks [42, 68, 70, 86]) suffer from the performance issue in general.

Table 4. Categorizing existing distributed transaction systems in terms of SNOW-Optimality, latency-optimality, and data consistency. "✓", resp. "×", refers to satisfying, resp. violating, the property. *Read atomic* transaction algorithms are colored in blue.

Dist. Database System	SNOW Optimality	Latency Optimality	Data Consistency
Spanner-Snap [24]	✓	✓	Serializability
Yesquel [2]	✓	✓	Snapshot Isolation
Eiger-PORT [55]	✓	✓	Transactional Causality
LORA	✓	✓	Read Atomicity
MySQL Cluster [60]	✓	✓	Read Committed
Committed Reads (CR)	✓	✓	Read Committed
DrTM [84]	✓	×	Strict Serializability
RIFL [40]	✓	×	Strict Serializability
Sinfonia [3]	✓	×	Strict Serializability
Spanner-RO [24]	✓	×	Strict Serializability
Rococo-SNOW [54]	✓	×	Strict Serializability
Algorithm B [37]	✓	×	Strict Serializability
Algorithm C [37]	✓	×	Strict Serializability
Scylla-PORT [55]	×	✓	Process-Ordered Serializability
COPS-SNOW [54]	×	✓	Transactional Causality
Pileus-Strong [78]	×	×	Strict Serializability
TAPIR [89]	×	×	Serializability
Walter [74]	×	×	Parallel Snapshot Isolation
Jessy [8]	×	×	Non-Monotonic Snapshot Isolation
ROLA [48]	×	×	Update Atomicity
Cure [4]	×	×	Transactional Causality
COPS [52]	×	×	Causal Consistency
RAMP-Fast (RF) [10]	×	×	Read Atomicity
RAMP-Small [10]	×	×	Read Atomicity
RAMP-Hybrid [10]	×	×	Read Atomicity
Faster Commit (FC) [10]	×	×	Read Atomicity
One-Phase Writes (1PW) [10]	×	×	Read Atomicity
RAMP-TAO [21]	×	×	Read Atomicity

9 RELATED WORK

9.1 SNOW-Optimal and Latency-Optimal Reads

Table 4 categorizes many recent distributed database systems in terms of SNOW-optimality, latency-optimality, and data consistency. Like LORA, Spanner-Snap [24], Yesquel [2], Eiger-PORT [55], MySQL Cluster [60], and Committed Reads are also SNOW-optimal and latency-optimal. However, instead of RA, they target different data consistency models, thus not suitable for web applications requiring exactly the RA guarantee, e.g., the "becoming friends" and "likes and liked by" associations in many social networking applications.

Many cloud database systems are either SNOW-optimal (DrTM [84], RIFL [40], Spanner-RO [24], Rococo-SNOW [54], etc.) or latency-optimal (such as Scylla-PORT [55] and COPS-SNOW [54]). As far as we know, no *read atomic* transaction algorithm exists in either category.

Most of existing distributed transaction algorithms are neither SNOW-optimal nor latency-optimal (which indicates potential improvements to their transactional reads). See Table 4 for a few examples where we only cite one transaction

algorithm for each data consistency guarantee. All existing *read atomic* transaction algorithms fall into this category. As discussed in Section 3.1 and demonstrated by the experimental results in Fig. 4, even the optimized RAMP designs, 1PW and FC, still require 1.5 round trips in average for transactional reads and underperform our LORA algorithm.

9.2 Formal Methods for Developing Cloud Databases

9.2.1 Maude-Based Approaches. The Maude ecosystem has been used to design, verify, and/or implement a number of distributed database systems.

Read Atomic Transactions. Regarding the RAMP-family algorithms, Maude's linear temporal logic model checker and PVeStA are employed to analyze the correctness properties such as RA and the performance properties such as transaction latency [45, 47]. On the system side, none of these algorithms considers SNOW-optimal reads, one-phase writes with RYW, or fully functional read-write transactions as in LORA. On the formal methods side, LORA's development follows a more general methodology where (i) correctness properties are mathematically proved within the same semantics framework, (ii) distributed implementations are generated with preserved correctness, and (iii) qualitative and quantitative properties are analyzed on both design and implementation level; in particular, evaluations under realistic deployments confirm the model-based statistical predications.

Other Data Consistency Guarantees. Maude, Real-Time Maude [65], and PVeStA have also been used to model and analyze distributed databases that provide other data consistency guarantees [34, 44, 48, 50, 64]. The paper [64] formally checks the *serializable* data store P-Store using reachability analysis without any quantitative measurement. The paper [34] model checks Google's Megastore against *serializability* and estimates the transaction latency using randomized Real-Time Maude simulations. Unlike SMC, this approach cannot offer any measure of statistical confidence in the estimations at the design stage.

The paper [48] presents a formal design of the ROLA transaction system in Maude and performs automated model checking and statistical verification analyses. However, only informal argument of ROLA's *update atomicity* is given based on a different formalism, which introduces a semantic gap. In contrast, within the same semantics framework, we formally prove LORA's consistency properties and perform automatic formal analysis.

The papers [44, 50] illustrate how the Cassandra key-value store (supporting single read/write operations) can be formally analyzed for both non-transactional data consistency and performance properties. Transactional consistency properties considered in this paper are more complex to be analyzed.

All the above formal efforts are restricted to the design level, while our approach passes the formal design of LORA to the actual distributed implementation where correctness testing and performance evaluation are performed. In a similar vein, the paper [51] also uses the D transformation to generate distributed Maude implementations for ROLA and a locking-based transaction protocol (providing *serializability*), and compares the implementation-based evaluations with the SMC-based estimations. Compared to our methodology and toolkit, no session guarantee is investigated; no formal proof of the consistency properties is given; and the system deployment is not fully automated.

The Maude ecosystem has also been successfully applied to analyze database-backed cloud services. To cite a few examples, Skeirik *et al.* [71] formally model and analyze availability properties of a ZooKeeper-based group key management service; Eckhardt *et al.* [31] propose and analyze DoS resilience mechanisms for Internet services; the constructor-based reachability logic theorem prover is used to verify the security properties of a browser system [72].

9.2.2 Other Formal Development Methods. Many frameworks and formalisms have been developed to model and verify distributed systems in general and cloud databases in particular.

Design-Level Verification and Code Extraction. TLA+ and its model checker TLC have been used in the development of distributed data stores in both industry, e.g., Amazon's cloud computing infrastructure [63] and Microsoft's Azure Cosmos DB [58], and academia, e.g., the TAPIR transaction protocol [89] and the Zeus distributed transactions [36].

The Chapar framework [42] is specialized to extract correct-by-construction OCaml implementations of causally consistent key-value stores from their formal specifications expressed and verified in Coq. Analogously, the frameworks Verdi [86], Disel [70], and Velisarios [68] specify both distributed system designs and desired properties in Coq, use Coq to prove the properties, and extract correct OCaml code.

PSync [30] is a domain-specific language embedded in Scala which is used to implement and verify fault-tolerant distributed algorithms. Both safety and liveness properties are checked over PSync programs and preserved in the runtime systems via refinement.

These formal methods do not support rapid exploration of system performance before the actual implementation and deployment. This is indeed one of the complaints by the Amazon engineers [63]. With our toolkit, not only the correctness properties can be model-checked and preserved in the distributed implementation, but also the performance can be predicted based on the formal system designs.

Many prevalent formal analysis tools, such as Uppaal [80] and Prism [38], are based on finite automata models. We are not aware of any work on formalizing and analyzing cloud databases with such tools. This might be because these object-based distributed systems have intrinsic features that are quite hard or impossible to represent in finite automata models. For instance, object attributes may contain unbounded data structures; asynchronous message passing and dynamic object creation may increase the number of both messages and objects in an unbounded manner. In contrast, we base our methodology on rewriting logic and Maude that provide the expressiveness and modeling convenience.

Code Verification and Hybrid Approaches. Both SAMC [41] and MODIST [87] employ distributed model checkers to verify the implementations of cloud systems. The main difficulty is that system properties in general (e.g., *read atomicity*) can be considerably harder or even impossible to express and prove at the imperative program level.

Verifying both formal design and actual implementation is the goal of the IronFleet framework [35]. The verification methodology includes a wide range of methods and tools (e.g., TLA+, Hoare logic assertions, the Dafny prover, and Z3) that require considerable assistance from the developers.

The Igloo framework [76] soundly links compositional refinement and separation logic for distributed system verification with I/O specifications encoded into separation logic assertions. This methodology only supports safety properties and the data consistency specifications in general cannot be checked using standard local assertions.

The P programming framework [26] provides language support for implementing and testing distributed systems. The built-in safety checker allows one to verify P programs which are then compiled to C code. A semantic gap is therefore introduced between the verified and the executable artifacts. ModP [27] extends P with compositional programming and testing, while the focus is to facilitate building more complex distributed systems.

Stateright [61] is a Rust-based framework for systematically verifying distributed systems. It can test all possible observable behaviors within a particular specification but needs to be embedded into the system's implementation beforehand. Correct embedding requires touching the large code base, which may be labor-intensive with complex distributed systems like cloud databases.

10 CONCLUDING REMARKS

We have presented LORA, the first SNOW-optimal *read atomic* transaction algorithm that also satisfies the *read your writes* session guarantee. We have formalized the LORA design and performed comprehensive formal analyses, including mathematical proofs, automated model checking, and statistical verification, with respect to both qualitative and quantitative properties. Moreover, we have obtained from our LORA model a correct-by-construction distributed implementation that has been deployed in a real cluster, tested for data consistency guarantees, and evaluated for system performance. Our experimental results have validated our proofs of LORA's correctness and demonstrated its promising performance and consistency achievement.

This work has shown, to the best of our knowledge for the first time, that a full development of a new distributed transaction system can be achieved within the same framework and with a single artifact by the use of formal methods. Our methodology is generic and (with some extensions) could be applied to develop provably correct, predictably high-performance distributed systems in general with automatically generated correct and deployable implementations.

In addition to the potential improvement to our toolkit (Section 8), we are also considering an orthogonal line of research on extending and optimizing LORA. The obvious next step is to extend LORA with data replication for fault tolerance. One option is to design and implement a replicated LORA algorithm with classic replication mechanisms such as *master-slave replication* [39] and *quorum-based voting* [32]. Thanks to the compatibility of reads and writes (the W property), database replicas would safely process different transactions in parallel. Alternatively, we could layer LORA on top of an existing replicated data store such as Cassandra [19] (similar to the RAMP-TAO approach [21]), which then requires a thorough design of APIs.

To refresh views of the latest committed writes LORA requires metadata size linear in the size of sibling keys. Transmitting more metadata would increase latency and decrease throughput. Hence, encoding metadata in a constant manner with recent novel data structures such as *version clocks* [55] seems a promising topic for future research.

Data Availability. The Maude specifications of the five read atomic transaction algorithms, the transformed probabilistic models, the generated distributed implementations, the extended consistency property library of CAT, the quantitative properties, the automated deploying tool, and the parametric workload generator are available at [43].

REFERENCES

[1] Gul A. Agha, José Meseguer, and Koushik Sen. 2006. PMaude: Rewrite-based Specification Language for Probabilistic Object Systems. *Electr. Notes Theor. Comput. Sci.* 153, 2 (2006).

[2] Marcos K. Aguilera, Joshua B. Leners, and Michael Walfish. 2015. Yesquel: scalable sql storage for web applications. In *SOSP*. ACM, 245–262.

[3] Marcos Kawazoe Aguilera, Arif Merchant, Mehul A. Shah, Alistair C. Veitch, and Christos T. Karamanolis. 2009. Sinfonia: A new paradigm for building scalable distributed systems. *ACM Trans. Comput. Syst.* 27, 3 (2009), 5:1–5:48.

[4] Deepthi Devaki Akkoorath, Alejandro Z. Tomsic, Manuel Bravo, Zhongmiao Li, Tyler Crain, Annette Bieniusa, Nuno M. Preguiça, and Marc Shapiro. 2016. Cure: Strong Semantics Meets High Availability and Low Latency. In *ICDCS'16*. IEEE Computer Society, 405–414.

[5] Musab AlTurki and José Meseguer. 2011. PVeStA: A Parallel Statistical Model Checking and Quantitative Analysis Tool. In *CALCO'11 (LNCS)*, Vol. 6859. Springer, 386–392.

[6] Rajeev Alur, Thomas A. Henzinger, and Moshe Y. Vardi. 2015. Theory in practice for system design and verification. *ACM SIGLOG News* 2, 1 (2015), 46–51.

[7] Eric Anderson, Xiaozhou Li, Mehul A. Shah, Joseph Tucek, and Jay J. Wylie. 2010. What Consistency Does Your Key-Value Store Actually Provide?. In *HotDep*. USENIX Association.

[8] Masoud Saeida Ardekani, Pierre Sutra, and Marc Shapiro. 2013. Non-monotonic Snapshot Isolation: Scalable and Strong Consistency for Geo-replicated Transactional Systems. In *SRDS*. 163–172.

[9] Peter Bailis, Aaron Davidson, Alan Fekete, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. 2013. Highly Available Transactions: Virtues and Limitations. *PVLDB* 7, 3 (2013).

[10] Peter Bailis, Alan Fekete, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. 2016. Scalable Atomic Visibility with RAMP Transactions. *ACM Trans. Database Syst.* 41, 3 (2016), 15:1–15:45.
[11] Theophilus Benson, Aditya Akella, and David A. Maltz. 2010. Network traffic characteristics of data centers in the wild. In *IMC*. ACM, 267–280.
[12] Hal Berenson, Philip A. Bernstein, Jim Gray, Jim Melton, Elizabeth J. O'Neil, and Patrick E. O'Neil. 1995. A Critique of ANSI SQL Isolation Levels. In *SIGMOD*. ACM, 1–10.
[13] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. 1987. *Concurrency Control and Recovery in Database Systems.* Addison Wesley.
[14] Yves Bertot and Pierre Castéran. 2004. *Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions.* Springer.
[15] Ranadeep Biswas and Constantin Enea. 2019. On the complexity of checking transactional consistency. *Proc. ACM Program. Lang.* 3, OOPSLA (2019), 165:1–165:28.
[16] Rakesh Bobba, Jon Grov, Indranil Gupta, Si Liu, José Meseguer, Peter Csaba Ölveczky, and Stephen Skeirik. 2018. Survivability: Design, Formal Modeling, and Validation of Cloud Storage Systems Using Maude. In *Assured Cloud Computing*. Wiley-IEEE Computer Society Press, Chapter 2, 10–48.
[17] Eric A. Brewer. 2000. Towards robust distributed systems (abstract). In *PODC*. 7.
[18] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry C. Li, Mark Marchukov, Dmitri Petrov, Lovro Puzar, Yee Jiun Song, and Venkateshwaran Venkataramani. 2013. TAO: Facebook's Distributed Data Store for the Social Graph. In *USENIX ATC'13*. USENIX Association, 49–60.
[19] Cassandra. 2021. http://cassandra.apache.org
[20] Andrea Cerone, Giovanni Bernardi, and Alexey Gotsman. 2015. A Framework for Transactional Consistency Models with Atomic Visibility. In *CONCUR*. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik.
[21] Audrey Cheng, Xiao Shi, Lu Pan, Anthony Simpson, Neil Wheaton, Shilpa Lawande, Nathan Bronson, Peter Bailis, Natacha Crooks, and Ion Stoica. 2021. RAMP-TAO: Layering Atomic Transactions on Facebook's Online TAO Data Store. *PVLDB* (2021).
[22] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn L. Talcott. 2007. *All About Maude.* LNCS, Vol. 4350. Springer.
[23] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *SOCC'10*. ACM, 143–154.
[24] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, et al. 2012. Spanner: Google's Globally-Distributed Database. In *OSDI 2012*. 261–264.
[25] James A. Cowling and Barbara Liskov. 2012. Granola: Low-Overhead Distributed Transaction Coordination. In *USENIX ATC*. USENIX Association, 223–235.
[26] Ankush Desai, Vivek Gupta, Ethan K. Jackson, Shaz Qadeer, Sriram K. Rajamani, and Damien Zufferey. 2013. P: safe asynchronous event-driven programming. In *PLDI*. ACM, 321–332.
[27] Ankush Desai, Amar Phanishayee, Shaz Qadeer, and Sanjit A. Seshia. 2018. Compositional programming and testing of dynamic distributed systems. *Proc. ACM Program. Lang.* 2, OOPSLA (2018), 159:1–159:30.
[28] Akon Dey, Alan Fekete, Raghunath Nambiar, and Uwe Röhm. 2014. YCSB+T: Benchmarking web-scale transactional databases. In *ICDE*. IEEE Computer Society, 223–230.
[29] Diego Didona, Rachid Guerraoui, Jingjing Wang, and Willy Zwaenepoel. 2018. Causal Consistency and Latency Optimality: Friend or Foe? *Proc. VLDB Endow.* 11, 11 (2018), 1618–1632.
[30] Cezara Dragoi, Thomas A. Henzinger, and Damien Zufferey. 2016. PSync: a partially synchronous language for fault-tolerant distributed algorithms. In *POPL*. ACM, 400–415.
[31] Jonas Eckhardt, Tobias Mühlbauer, Musab AlTurki, José Meseguer, and Martin Wirsing. 2012. Stable Availability under Denial of Service Attacks through Formal Patterns. In *FASE*. 78–93.
[32] David K. Gifford. 1979. Weighted Voting for Replicated Data. In *SOSP*. ACM, 150–162.
[33] Wojciech Golab, Xiaozhou Li, and Mehul A. Shah. 2011. Analyzing consistency properties for fun and profit. In *PODC'11*. ACM, 197–206.
[34] Jon Grov and Peter Csaba Ölveczky. 2014. Formal Modeling and Analysis of Google's Megastore in Real-Time Maude. In *Specification, Algebra, and Software (LNCS)*, Vol. 8373. Springer.
[35] C. Hawblitzel, J. Howell, M. Kapritsos, J. R. Lorch, B. Parno, M. L. Roberts, S. T. V. Setty, and B. Zill. 2015. IronFleet: proving practical distributed systems correct. In *SOSP*. ACM.
[36] Antonios Katsarakis, Yijun Ma, Zhaowei Tan, Andrew Bainbridge, Matthew Balkwill, Aleksandar Dragojevic, Boris Grot, Bozidar Radunovic, and Yongguang Zhang. 2021. Zeus: locality-aware distributed transactions. In *EuroSys*. ACM, 145–161.
[37] Kishori M. Konwar, Wyatt Lloyd, Haonan Lu, and Nancy A. Lynch. 2021. SNOW Revisited: Understanding When Ideal READ Transactions Are Possible. In *IPDPS 2021*. IEEE, 922–931.
[38] M. Kwiatkowska, G. Norman, and D. Parker. 2011. PRISM 4.0: Verification of Probabilistic Real-time Systems. In *CAV'11 (LNCS)*, Vol. 6806. Springer, 585–591.
[39] Rivka Ladin, Barbara Liskov, Liuba Shrira, and Sanjay Ghemawat. 1992. Providing High Availability Using Lazy Replication. *ACM Trans. Comput. Syst.* 10, 4 (1992), 360–391.

[40] Collin Lee, Seo Jin Park, Ankita Kejriwal, Satoshi Matsushita, and John K. Ousterhout. 2015. Implementing linearizability at large scale and low latency. In *SOSP*. ACM, 71–86.

[41] T. Leesatapornwongsa, M. Hao, P. Joshi, J. F. Lukman, and H. S. Gunawi. 2014. SAMC: Semantic-Aware Model Checking for Fast Discovery of Deep Bugs in Cloud Systems. In *OSDI*. USENIX Association.

[42] Mohsen Lesani, Christian J. Bell, and Adam Chlipala. 2016. Chapar: certified causally consistent distributed key-value stores. In *POPL'16*. ACM, 357–370.

[43] Si Liu. 2021. All in One: Design, Verification, and Implementation of SNOW-Optimal Read Atomic Transactions. Artifacts. https://github.com/siliunobi/lora.

[44] Si Liu, Jatin Ganhotra, Muntasir Rahman, Son Nguyen, Indranil Gupta, and José Meseguer. 2017. Quantitative Analysis of Consistency in NoSQL Key-Value Stores. *Leibniz Transactions on Embedded Systems* 4, 1 (2017), 03:1–03:26.

[45] Si Liu, Peter Csaba Ölveczky, Jatin Ganhotra, Indranil Gupta, and José Meseguer. 2017. Exploring Design Alternatives for RAMP Transactions through Statistical Model Checking. In *ICFEM (LNCS)*. Springer.

[46] Si Liu, Peter Csaba Ölveczky, and José Meseguer. 2016. Modeling and analyzing mobile ad hoc networks in Real-Time Maude. *J. Log. Algebr. Meth. Program.* 85, 1 (2016), 34–66.

[47] Si Liu, Peter Csaba Ölveczky, Muntasir Raihan Rahman, Jatin Ganhotra, Indranil Gupta, and José Meseguer. 2016. Formal modeling and analysis of RAMP transaction systems. In *SAC*. ACM.

[48] Si Liu, Peter Csaba Ölveczky, Qi Wang, Indranil Gupta, and José Meseguer. 2019. Read atomic transactions with prevention of lost updates: ROLA and its formal analysis. *Formal Asp. Comput.* 31, 5 (2019), 503–540.

[49] Si Liu, Peter Csaba Ölveczky, Min Zhang, Qi Wang, and José Meseguer. 2019. Automatic Analysis of Consistency Properties of Distributed Transaction Systems in Maude. In *TACAS'19 (LNCS)*, Vol. 11428. Springer, 40–57.

[50] Si Liu, Muntasir Raihan Rahman, Stephen Skeirik, Indranil Gupta, and José Meseguer. 2014. Formal Modeling and Analysis of Cassandra in Maude. In *ICFEM (LNCS)*, Vol. 8829. Springer.

[51] Si Liu, Atul Sandur, José Meseguer, Peter Csaba Ölveczky, and Qi Wang. 2020. Generating Correct-by-Construction Distributed Implementations from Formal Maude Designs. In *NFM'20 (LNCS)*, Vol. 12229. Springer.

[52] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. 2011. Don't settle for eventual: scalable causal consistency for wide-area storage with COPS. In *SOSP 2011*. ACM, 401–416.

[53] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. 2013. Stronger Semantics for Low-Latency Geo-Replicated Storage. In *NSDI*. USENIX Association, 313–328.

[54] Haonan Lu, Christopher Hodsdon, Khiem Ngo, Shuai Mu, and Wyatt Lloyd. 2016. The SNOW Theorem and Latency-optimal Read-only Transactions. In *OSDI'16*. USENIX Association, 135–150.

[55] Haonan Lu, Siddhartha Sen, and Wyatt Lloyd. 2020. Performance-Optimal Read-Only Transactions. In *OSDI*. USENIX Association, 333–349.

[56] José Meseguer. 1992. Conditional Rewriting Logic as a Unified Model of Concurrency. *Theoretical Computer Science* 96, 1 (1992), 73–155.

[57] José Meseguer. 1993. A Logical Theory of Concurrent Objects and its realization in the Maude Language. In *Research Directions in Concurrent Object-Oriented Programming*, Gul Agha, Peter Wegner, and Akinori Yonezawa (Eds.). MIT Press, 314–390.

[58] Microsoft. 2018. High-level TLA+ specifications for the five consistency levels offered by Azure Cosmos DB. https://github.com/Azure/azure-cosmos-tla.

[59] Shuai Mu, Yang Cui, Yang Zhang, Wyatt Lloyd, and Jinyang Li. 2014. Extracting More Concurrency from Distributed Transactions. In *OSDI*. USENIX Association, 479–494.

[60] MySQL. 2021. MySQL Cluster CGE. https://www.mysql.com/products/cluster/.

[61] Jonathan Nadal. 2021. Stateright Actor Framework. https://www.stateright.rs.

[62] C. Newcombe, T. Rath, F. Zhang, B. Munteanu, M. Brooker, and M. Deardeuff. 2015. How Amazon Web Services Uses Formal Methods. *Commun. ACM* 58, 4 (April 2015), 66–73.

[63] Chris Newcombe, Tim Rath, Fan Zhang, Bogdan Munteanu, Mark Brooker, and Michael Deardeuff. 2015. How Amazon Web Services Uses Formal Methods. *Commun. ACM* 58, 4 (2015), 66–73.

[64] Peter Csaba Ölveczky. 2016. Formalizing and Validating the P-Store Replicated Data Store in Maude. In *WADT'16 (LNCS)*, Vol. 10644. Springer, 189–207.

[65] Peter Csaba Ölveczky and José Meseguer. 2008. The Real-Time Maude Tool. In *TACAS (LNCS)*, Vol. 4963. Springer, 332–336.

[66] Christos H. Papadimitriou. 1979. The Serializability of Concurrent Database Updates. *J. ACM* 26, 4 (Oct. 1979), 631–653.

[67] Lin Qiao, Kapil Surlaker, Shirshanka Das, Tom Quiggle, et al. 2013. On brewing fresh espresso: LinkedIn's distributed data serving platform. In *SIGMOD*. ACM, 1135–1146.

[68] Vincent Rahli, Ivana Vukotic, Marcus Völp, and Paulo Jorge Esteves Veríssimo. 2018. Velisarios: Byzantine Fault-Tolerant Protocols Powered by Coq. In *ESOP (LNCS)*, Vol. 10801. Springer, 619–650.

[69] Koushik Sen, Mahesh Viswanathan, and Gul Agha. 2005. On Statistical Model Checking of Stochastic Systems. In *CAV'05 (LNCS)*, Vol. 3576. Springer.

[70] Ilya Sergey, James R. Wilcox, and Zachary Tatlock. 2017. Programming and Proving with Distributed Protocols. *Proc. ACM Program. Lang.* 2, POPL, Article 28 (2017), 28:1–28:30 pages.

[71] Stephen Skeirik, Rakesh B. Bobba, and José Meseguer. 2013. Formal Analysis of Fault-tolerant Group Key Management Using ZooKeeper. In *CCGRID*. 636–641.
[72] Stephen Skeirik, José Meseguer, and Camilo Rocha. 2020. Verification of the IBOS Browser Security Properties in Reachability Logic. In *WRLA (LNCS)*, Vol. 12328. Springer, 176–196.
[73] Stephen Skeirik, Andrei Stefanescu, and José Meseguer. 2017. A Constructor-Based Reachability Logic for Rewrite Theories. In *LOPSTR (LNCS)*, Vol. 10855. Springer, 201–217.
[74] Yair Sovran, Russell Power, Marcos K. Aguilera, and Jinyang Li. 2011. Transactional storage for geo-replicated systems. In *SOSP'11*. ACM, 385–400.
[75] Kristina Spirovska, Diego Didona, and Willy Zwaenepoel. 2019. PaRiS: Causally Consistent Transactions with Non-blocking Reads and Partial Replication. In *ICDCS'19*. IEEE, 304–316.
[76] Christoph Sprenger, Tobias Klenze, Marco Eilers, Felix A. Wolf, Peter Müller, Martin Clochard, and David A. Basin. 2020. Igloo: soundly linking compositional refinement and separation logic for distributed system verification. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 152:1–152:31.
[77] Douglas B. Terry, Alan J. Demers, Karin Petersen, Mike Spreitzer, Marvin Theimer, and Brent B. Welch. 1994. Session Guarantees for Weakly Consistent Replicated Data. In *PDIS*. IEEE Computer Society, 140–149.
[78] Douglas B. Terry, Vijayan Prabhakaran, Ramakrishna Kotla, Mahesh Balakrishnan, Marcos K. Aguilera, and Hussam Abu-Libdeh. 2013. Consistency-based Service Level Agreements for Cloud Storage. In *SOSP*. ACM, 309–324.
[79] Alejandro Z. Tomsic, Manuel Bravo, and Marc Shapiro. 2018. Distributed transactional reads: the strong, the quick, the fresh & the impossible. In *Middleware*. ACM, 120–133.
[80] Uppaal. 2019. Uppaal 4.0.15. http://www.uppaal.org.
[81] Hiroshi Wada, Alan Fekete, Liang Zhao, Kevin Lee, and Anna Liu. 2011. Data Consistency Properties and the Trade-offs in Commercial Cloud Storage: the Consumers' Perspective. In *CIDR*. 134–143.
[82] Anduo Wang, Alexander J. T. Gurney, Xianglong Han, Jinyan Cao, Boon Thau Loo, Carolyn L. Talcott, and Andre Scedrov. 2014. A reduction-based approach towards scaling up formal analysis of internet configurations. In *INFOCOM*. IEEE, 637–645.
[83] Qi Wang, Pubali Datta, Wei Yang, Si Liu, Adam Bates, and Carl A. Gunter. 2019. Charting the Attack Surface of Trigger-Action IoT Platforms. In *CCS*. ACM, 1439–1453.
[84] Xingda Wei, Jiaxin Shi, Yanzhe Chen, Rong Chen, and Haibo Chen. 2015. Fast in-memory transaction processing using RDMA and HTM. In *SOSP*. ACM, 87–104.
[85] Brian White, Jay Lepreau, Leigh Stoller, Robert Ricci, Shashi Guruprasad, Mac Newbold, Mike Hibler, Chad Barb, and Abhijeet Joglekar. 2002. An Integrated Experimental Environment for Distributed Systems and Networks. In *OSDI*. USENIX Association.
[86] James R. Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Thomas Anderson. 2015. Verdi: A Framework for Implementing and Formally Verifying Distributed Systems. In *PLDI'15*. ACM, 357–368.
[87] J. Yang, T. Chen, M. Wu, Z. Xu, X. Liu, H. Lin, M. Yang, F. Long, L. Zhang, and L. Zhou. 2009. MODIST: Transparent Model Checking of Unmodified Distributed Systems. In *NSDI*. USENIX Association, 213–228.
[88] Håkan L. S. Younes and Reid G. Simmons. 2006. Statistical probabilistic model checking with a focus on time-bounded properties. *Inf. Comput.* 204, 9 (2006), 1368–1409.
[89] Irene Zhang, Naveen Kr. Sharma, Adriana Szekeres, Arvind Krishnamurthy, and Dan R. K. Ports. 2015. Building consistent transactions with inconsistent replication. In *SOSP*. ACM, 263–278.

A THE RAMP-FAST ALGORITHM

Fig. 5 shows the pseudo-code of the RAMP-Fast algorithm as given in [10].

ALGORITHM 1: RAMP-Fast

Server-side Data Structures

1: *versions*: set of versions $\langle item, value$, timestamp ts_v, metadata $md\rangle$
2: $lastCommit[i]$: last committed timestamp for item i

Server-side Methods

3: **procedure** PREPARE(v : version)
4: $versions$.add(v)
5: **return**
6: **procedure** COMMIT(ts_c : timestamp)
7: $I_{ts} \leftarrow \{w.item \mid w \in versions \wedge w.ts_v = ts_c\}$
8: $\forall i \in I_{ts}, lastCommit[i] \leftarrow \max(lastCommit[i], ts_c)$
9: **procedure** GET(i : item, ts_{req} : timestamp)
10: **if** $ts_{req} = \emptyset$ **then**
11: **return** $v \in versions : v.item = i \wedge v.ts_v = lastCommit[item]$
12: **else**
13: **return** $v \in versions : v.item = i \wedge v.ts_v = ts_{req}$

Client-side Methods

14: **procedure** PUT_ALL(W : set of $\langle item, value\rangle$)
15: $ts_{tx} \leftarrow$ generate new timestamp
16: $I_{tx} \leftarrow$ set of items in W
17: **parallel-for** $\langle i, v\rangle \in W$
18: $w \leftarrow \langle item = i, value = v, ts_v = ts_{tx}, md = (I_{tx} - \{i\})\rangle$
19: invoke PREPARE(w) on respective server (i.e., partition)
20: **parallel-for** server s : s contains an item in W
21: invoke COMMIT(ts_{tx}) on s
22: **procedure** GET_ALL(I : set of items)
23: $ret \leftarrow \{\}$
24: **parallel-for** $i \in I$
25: $ret[i] \leftarrow$ GET($i, \emptyset$)
26: $v_{latest} \leftarrow \{\}$ (default value: -1)
27: **for** response $r \in ret$ **do**
28: **for** $i_{tx} \in r.md$ **do**
29: $v_{latest}[i_{tx}] \leftarrow \max(v_{latest}[i_{tx}], r.ts_v)$
30: **parallel-for** item $i \in I$
31: **if** $v_{latest}[i] > ret[i].ts_v$ **then**
32: $ret[i] \leftarrow$ GET($i, v_{latest}[i]$)
33: **return** ret

Fig. 5. The pseudo-code of RAMP-Fast as given in [10].

B COMPARISON BETWEEN LORA AND 1PW

Fig. 6 shows an example scenario that distinguishes LORA and 1PW with respect to *read your writes*: with transactional writes committed in one round trip, LORA satisfies RYW while 1PW violates.

Example B.1 (Fig. 6). Consider a scenario of two racing transactions, $T_1 : [w(x_1), w(y_1)]$ and $T_2 : [r(x), r(y)]$, issued in the same client session, where T_2 starts immediately after T_1 completes the PREPARE phase and commits before

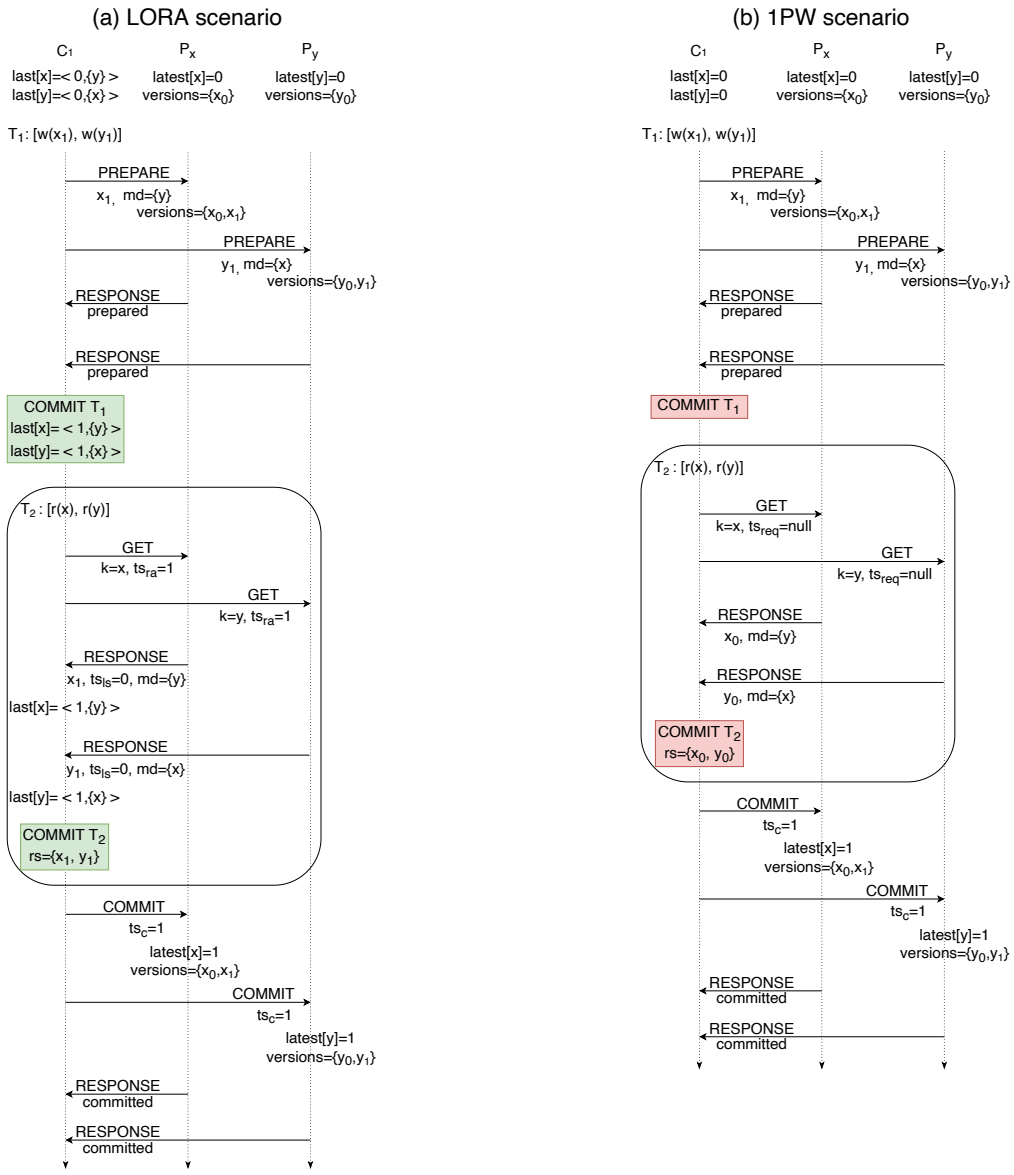

Fig. 6. Scenarios illustrating the difference between LORA and 1PW with respect to RYW: reading $\{x_1, y_1\}$ with LORA satisfies RYW while returning $\{x_0, y_0\}$ with 1PW fails to.

T_1 issues the COMMIT phase. A LORA coordinator C_1 incorporates T_1's writes into the snapshot (i.e., $ts_{ra} = 1$) for T_2. The partitions P_x and P_y then reply with the specified versions x_1 and y_1 that satisfy both RA and RYW. With 1PW coordinator C_1 issues T_2's reads with the *null* timestamp (as in RAMP-Fast) and fetches the latest committed versions x_0 and y_0 that satisfy RA but violate RYW.

C RULES FOR LORA'S WRITE-ONLY AND READ-WRITE TRANSACTIONS

The following presents the remaining 8 rewrite rules for LORA's write-only and read-write transactions. For each rule, we also refer to the corresponding lines of code in Algorithm 1.

C.1 Starting a Transaction

A coordinator starts to execute a transaction (independent of its type) by moving the first transaction in queue to `executing`, if it is currently executing no transaction (indicated by `null`).

```
rl [start-txn] :    *** auxiliary rule for starting a txn
   < C : Coord | queue : < T : Txn | > TRANSES, executing : null >
 =>
   < C : Coord | queue : TRANSES, executing : < T : Txn | > > .
```

C.2 LORA's Write-Only Transactions

Issuing PUT Messages (Lines 9–15). A coordinator executes a write-only transaction (ensured by the `writeOnly` predicate) if `executing` holds the transaction object in the left-hand side of a rule.

```
crl [put-all] :
 < C : Coord | executing : < T : Txn | waitinglist : empty, writeset : empty, operations : OPS >,
               sqn : SQN, mapping : KP >
=>
 < C : Coord | executing : < T : Txn | waitinglist : addW(OPS,KP) >, writeset : ws(OPS) >,
               sqn : SQN + 1 >
 putAll(C,T,OPS,SQN + 1,KP)    if writeOnly(OPS) .
```

The coordinator adds to the waiting list the partitions from which it awaits the votes (by the function `addW`). The PUT messages are generated by the `putAll` function (similarly defined as for function `getAll` in rule `[get-all]`; see [43] for its definition).

Receiving a PREPARE Message (Lines 31–33). Upon receiving the prepared version, the partition stores it in the local `database` and replies with a `prepared` vote.

```
rl [rcv-prepare] :
   (to P from C : prepare(T,K,VAL,TS,MD))
   < P : Partition | database : VS >
=>
   < P : Partition | database : VS ; < K,VAL,TS,MD > >
   (from P to C : prepared(T)) .
```

Receiving a Vote (Lines 16–19). Once the coordinator collects all votes (indicated by the `empty` waiting list `OS`), it commits the transaction by moving the transaction object to `committed` and incorporates the transactional writes into its local view by updating `last`. The coordinator also notifies each partition to commit the writes in its local database (by the COMMIT messages produced by function `commitAll`). The same set of partitions as in the PREPARE phase (rule `[put-all]`) is then stored in the waiting list (the `if-then` branch).

```
rl [rcv-prepared] :
```

```
  (to C from P : prepared(T))
  < C : Coord | executing : < T : Txn | operations : OPS, waitinglist : P ; OS >,
                last : LAS, sqn : SQN, committed : TXNS >
=>
  if OS == empty
    then < C : Coord | executing : null,
                       last : update(OPS,C,SQN,LAS),
                       committed : TXNS < T : Txn | waitinglist : addW(OPS,KP) > >
         commitAll(C,T,OPS,SQN,KP)
    else < C : Coord | executing : < T : Txn | waitinglist : OS > > fi .
```

Otherwise, the coordinator continues to await the remaining votes (the `else` branch).

Committing a Version (Lines 34–36). The partition marks the corresponding version (indicated by timestamp `TS`) as committed by updating the `latest` committed timestamp of the associated key.

```
rl [rcv-commit] :
   (to P from C : commit(T,TS))
   < P : Partition | database : VS, latest : LAT >
=>
   < P : Partition | latest : update(TS,VS,LAT) >
   (from P to C : committed(T)) .
```

Receiving an Ack. Upon receiving a `committed` message from some partition, the coordinator updates the waiting list accordingly.

```
rl [rcv-committed] :    *** on receiving an ack from a partition
   (to C from P : committed(T))
   < C : Coord | committed : TXNS < T : Txn | waitinglist : P ; OS > >
=>
   < C : Coord | committed : TXNS < T : Txn | waitinglist : OS > .
```

C.3 LORA's Read-Write Transactions

Invoking the UPDATE Method (Lines 20–21). LORA executes a read-write transaction (ensured by the predicate `readWrite`) by first fetching the versions of the requested keys. The corresponding `get` messages are generated by the `getAll` function (as in rule `[get-all]`; see Section 5.2). The key-specific partitions are also added to the waiting list.

```
crl [update] :
    < C : Coord | executing : < T : Txn | waitinglist : empty, readset : empty,
                                          operations : OPS >,
                  last : LAS, mapping : KP >
 =>
    < C : Coord | executing : < T : Txn | waitinglist : addR(OPS,KP) > >
    getAll(C,T,OPS,LAS,KP)    if readWrite(OPS) .
```

Finishing Reads (Lines 22–24). Upon completing the only round of reads, the coordinator proceeds with the transactional writes by issuing the `prepare` messages to the partitions, which are generated by the `putAll` function (as in rule `[put-all]`).

```
crl [finished-rs] :
    < C : Coord | executing : < T : Txn | operations : OPS, waitinglist : empty,
                                          readset : R ; RS, writeset : empty >,
                  sqn : SQN, mapping : KP >
 =>
    < C : Coord | executing : < T : Txn | waitinglist : addW(OPS,KP), writeset : ws(OPS) >,
                  sqn : SQN + 1 >
    putAll(C,T,OPS,SQN + 1,KP)    if readWrite(OPS) .
```

D PROOF FOR LORA'S RYW SESSION GUARANTEE

In this section we formally prove LORA's *read your writes* (RYW) session guarantee. In addition to the assumptions and notations in Section 6, we also use the following notations in our proof.

Notations. A transaction T issued by coordinator C is denoted by T_c. Likewise, a transactional read r, resp. write w, by coordinator C is written as r_c, resp. w_c. We use r_c^T, resp. w_c^T, to denote a read r, resp. write w, of transaction T_c.

The consistency property *read your writes* (RYW) [77] requires that a write's effect must be visible to its subsequent reads in the same session (i.e., by the same coordinator in our case). More specifically, to satisfy RYW, a read must return its previous write (on the same key) or a later write by a different coordinator. In the case of returning the initial version, the read must guarantee no previous write issued in the same session.

Definition D.1. For any transactional read r_c, it satisfies RYW if the returned version v_k is

(i) the initial version $\perp_k$, with no version v'_k written by a previous transactional write w_c;
(ii) ordered before v'_k, i.e., $v_k < v'_k$, which is also issued by coordinator C but before r_c; *or*
(iii) written by another coordinator C', where no transactional write exists before r_c in the same coordinator session; otherwise, for any such write w_c that writes v'_k, $v_k > v'_k$.

Theorem D.2. *LORA provides the read your writes session guarantee.*

Proof. We prove this theorem by contradiction and consider three cases according to Definition D.1.

Case 1. Suppose that a read r_c^T returns the initial version $\perp_k$ and there exists a version v_k written by $w_c^{T'}$ with T' issued before T by coordinator C (i.e., initially appearing before T in queue). Hence, coordinator C has already incorporated the writes of T' into its local view, i.e., `last` maps key k to version v, before T starts (rule `[rcv-prepared]`). Returning $\perp_k$ (i.e., the associated timestamp returned by the `max` function) means $\perp_k > v_k$, a contradiction.

Case 2. Suppose that a read r_c^T returns version v_k written by $w_c^{T_1}$ and there exists a version v'_k written by $w_c^{T_2}$ with T_2 issued after T_1 but before T. Thus, we have $v_k < v'_k$ due to the increased sequence number (rule `[put-all]` and rule `[finished-rs]`). Returning v_k (by the `max` function) means $v_k > v'_k$, a contradiction.

Case 3. Suppose that a read r_c^T returns version v_k written by $w_{c'}^{T_1}$ ($c \neq c'$) and there exists a version v'_k written by w_c^{T2} with $v'_k > v_k$ and T_2 issued before T. Hence, w_c^{T2} is visible to r_c^T, i.e., v'_k has been incorporated into coordinatorr C's local view before T starts (rule `[rcv-prepared]`). Returning v_k, instead of v'_k, indicates $v_k > v'_k$, a contradiction. The

case of no existing write issued before r_c^T by coordinator C is trivial as returning any write, including v_k written by $w_{c'}^{T_1}$, satisfies RYW. □

E COMPARING LORA WITH THE ALTERNATIVE DESIGN

Fig. 7 plots the SMC-based performance comparison between LORA and the alternative design ALT: LORA outperforms ALT under all workloads.

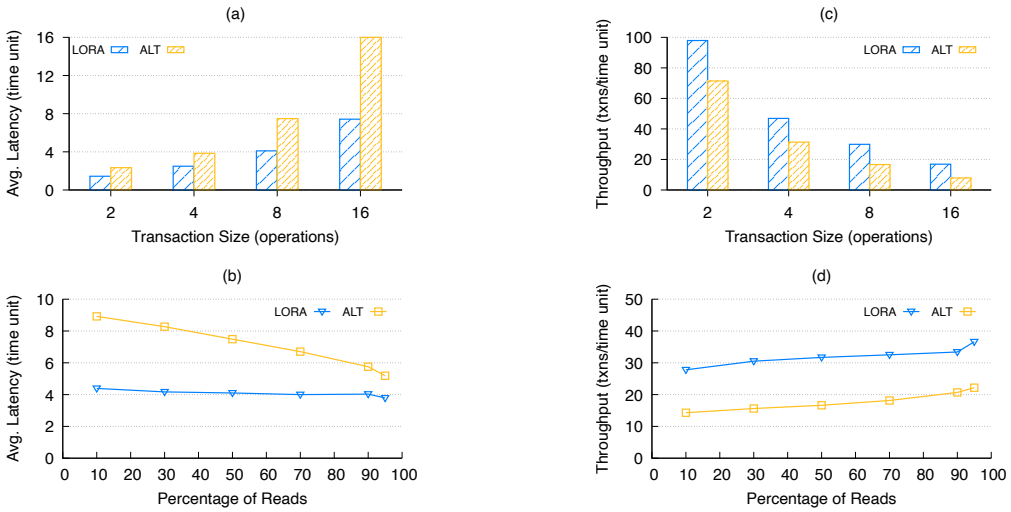

Fig. 7. SMC-based performance comparison between LORA and ALT.