Speculative Transaction Processing in Geo-Replicated Data Stores

Zhongmiao Li[†⋆], Peter Van Roy[†] and Paolo Romano[⋆]
[†]Université catholique de Louvain [⋆]Instituto Superior Técnico, Lisboa & INESC-ID

February 2017

Abstract

This work presents STR, a geo-distributed, partially replicated transactional data store, which leverages on novel speculative techniques to mask the inter-replica synchronization latency.

The theoretical foundations on top of which we built STR is a novel consistency criterion, which we call SPeculative Snapshot Isolation (SPSI). SPSI extends the well-known Snapshot Isolation semantics in an intuitive, yet rigorous way, by specifying desirable atomicity and isolation guarantees that shelter applications from subtle anomalies that can arise when adopting speculative transaction processing techniques.

We assess STR's performance on up to nine geo-distributed Amazon EC2 data centers, using both synthetic benchmarks as well as complex benchmarks (TPC-C and RUBiS). Our experimental study highlights that STR achieves throughput gains of up to 6× and latency reduction up to 100×, in workloads characterized by low inter-data center contention. Furthermore, thanks to self-tuning techniques that automatically adjust the aggressiveness of STR's speculation degree, STR offers robust performance even when faced with unfavourable workloads that suffer from high misspeculation rates.

Speculative Transaction Processing in Geo-Replicated Data Stores

Zhongmiao Li[†⋆]*, Peter Van Roy*[†] *and Paolo Romano*[⋆]
[†]*Université catholique de Louvain* [⋆]*Instituto Superior Técnico, Lisboa & INESC-ID*

1 Introduction

Modern online services are increasingly deployed over geographically-scattered data centers (geo-replication) [12, 27, 29]. Geo-replication allows services to remain available even in the presence of outages affecting entire data centers and it reduces access latency by bringing data closer to clients. On the down side, though, the performance of geographically distributed data stores is challenged by large communication delays between data centers. To provide ACID transactions, a desirable feature that can greatly simplify applications' development [41], some form of global certification is unavoidable in order to safely detect conflicts developed among concurrent transactions executing at different data centers. The adverse performance impact of inter-data center certification is of a twofold nature: i) system's throughput can be severely impaired, as transactions need to hold pre-commit locks during their global certification phase, which can cripple the effective concurrency that these systems can achieve; ii) client-perceived latency is also directly affected, since the inter-data center certification lies in the critical path of execution of transactions.

This work investigates the opportunities and challenges associated with the use of *speculative* processing techniques in geo-distributed partially replicated transactional data stores that provide a widely employed consistency criterion, i.e., Snapshot Isolation [13, 16] (SI). We focus on two speculative processing techniques, which we call: *speculative reads* and *speculative commits*.

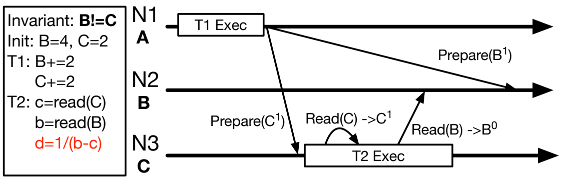
Speculative reads allow transactions to observe the data item versions produced by pre-committed transactions, instead of blocking until they are committed/aborted. As such, speculative reads can reduce the "effective duration" of pre-commit locks (i.e., as perceived by conflicting transactions), thus reducing transaction execution time and enhancing the maximum degree of parallelism achievable by the system — and, ultimately, throughput. We say that speculative reads are an *internal speculation* technique, as misspeculations caused by it never surface to the clients and can be dealt with by simply re-executing the affected transaction.

Speculative commits, instead, allow for exposing to external clients the results produced by transactions that are still undergoing their global certification phase. By removing the global certification phase from the critical path of transaction execution, speculative commits can drastically reduce the user-perceived latency. However, analogously to other techniques [33, 21] that externalize uncommitted state to clients — and that we call *external speculation* techniques — speculative commits require programmers to define compensation logic to deal explicitly with misspeculation events.

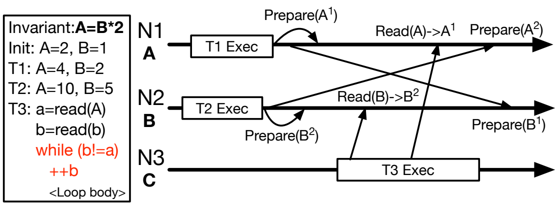
A number of works have already demonstrated how the use of speculative reads and speculative commits, either individually [18, 33, 25, 21] or in synergy [43], can significantly enhance the performance of distributed [43, 35, 25, 33, 34] and single-site [18] transactional systems. However, existing approaches suffer from several relevant limitations which represent the key motivation underlying the work presented in this paper:

1. Unfit for geo-distribution/partial replication. Some existing works in this area [35, 25, 43] were not designed for partially replicated geo-replicated data stores. On the contrary, they target different data models (i.e., full replication [35, 43]) or rely on techniques that impose prohibitive costs in WAN environments, such as the use of centralized sequencers to totally order transactions [25].

2. Subtle concurrency anomalies. To the best of our knowledge, all partially replicated geo-distributed transactional data stores that allow speculative reads [18, 22, 34] expose applications to anomalies that do not arise in non-speculative systems and that can severely undermine application correctness. Figure 1 illustrates two examples of concurrency anomalies that may arise when using these systems. The root cause of the problem is that existing systems allow speculative reads to observe *any* pre-committed data version, which exposes applications to observe data snapshots that reflect only partially the updates of transactions (Fig. 1a) and/or include versions created by conflicting concurrent transactions (Fig. 1b). These concurrency anomalies have the following negative impacts: 1) transaction execution may be affected to the extent that they



(a) Atomicity violation — T2 observes T1's pre-committed version of data item C, but not of B. This breaks the application invariant (B!=C), causing an unexpected division by zero exception that could crash the application at node N3.



(b) Isolation violation — T3 observes the pre-committed updates of two conflicting transactions, namely T1 and T2. T3 enters an infinite loop, as the application invariant (A=B*2) is broken due to the concurrency anomaly.

Figure 1: Examples illustrating possible concurrency anomalies caused by speculative reads. N1, N2 and N3 are three nodes that store data items A, B and C, respectively.

cause application's crashes or hangs without ever requesting to commit, and 2) if jointly used with speculative reads, speculative commits can observe and externalize non-atomic/non-isolated snapshots to human users or third-party applications. **3. Performance robustness.** If used injudiciously, speculation can hamper, instead of benefiting performance. As we will show, in adverse scenarios (e.g., large likelihood of transaction aborts and high system load) misspeculations can significantly penalize both user-perceived latency and system throughput.

This work aims to address precisely these limitations by introducing STR (Speculative Transaction Replication), a novel speculative transactional protocol for partially replicated geo-distributed data stores.

STR's shares several key design choices with state-of-the-art strongly consistent data stores [12, 13, 36], which contribute to its efficiency and scalability. These include: multi-versioning, which maximizes efficiency in read-dominated workloads [9], purely decentralized concurrency control based on distributed clocks [12, 13, 37], and support for partial replication [26, 12]. The key contribution of STR lies in its innovative distributed concurrency control scheme that supports speculative execution while providing intuitive and stringent consistency guarantees.

The theoretical foundations over which we built STR is a novel consistency model, which we called SPeculative Snapshot Isolation (SPSI) (§5). Besides guaranteeing the familiar Snapshot Isolation (SI) to *committed transactions*, SPSI provides clear and stringent guarantees on the atomicity and isolation of the snapshots observed and produced by *executing transactions* that use both speculative reads and speculative commits. In a nutshell, SPSI allows an executing transaction to not only read data item versions committed before it started (as in SI), but also to observe, in an atomic way, the effects of non-conflicting transactions that originated on the same node and pre-committed before it started.

Further, STR integrates a lightweight, yet effective, hill climbing-based self-tuning mechanism that dynamically adjusts the aggressiveness of the speculative mechanisms employed by the system based on the workload characteristics (§6). The use of self-tuning spares developers from the complexity of manually tuning any additional system knobs, by automatically identifying the configurations that maximize the performance gains achievable via speculation in favourable workloads, while ensuring robust performance in adverse scenarios that are unfavourable to the use of speculative techniques.

We evaluated STR on up to nine geo-distributed Amazon EC2 data centers, assessing its performance via both synthetic and complex benchmarks (TPC-C [4] and RUBiS [2]). Our experimental study highlighted that the use of speculative reads allows achieving up to 6× throughput improvements (in a completely transparent way for programmers) and that applications that exploit also speculative commits can achieve a further reduction of the user-perceived latency by up to 100×.

2 Related Work

Geo-replication. The problem of designing efficient mechanisms to ensure strong consistency semantics in geo-replicated data stores has been intensively studied. A class of geo-replicated systems [45, 14] is based on the *state-machine replication* (SMR) [28] approach, in which replicas first agree on the serialization order of transactions and then execute them without further coordination. Other recent systems [12, 13, 27, 30] adopt the *deferred update* (DU) [24] approach, in which transactions are first locally executed and then globally certified. This approach is more scalable than SMR in update intensive workloads [47, 24] and, unlike SMR, it can seamlessly support non-deterministic transactions [38]. The key down side of the DU approach is that locks must be maintained for the whole duration of transactions' global certification, which can severely hinder throughput [44]. STR builds on the DU approach and tackles its performance limitation via the use of speculative techniques.

Speculation. To mask latency in replicated systems, Helland et. al. advocate the *guesses and apologies* programming paradigm [23], in which systems expose preliminary results of requests (*guesses*), but reconcile the exposed results if they are different from final results (*apologies*). This corresponds to STR's notion of speculative commits, which is a programming approach adopted also in other recent systems, like PLANET [33] and ICG [21]. Unlike STR, though, these systems are designed to operate on conventional storage systems, which do not support speculative reads of pre-committed data. As such, these approaches can benefit user-perceived latency, but they do not tackle the problem of reducing transaction's blocking time, which can severely impair throughput. In fact, as we will show in our evaluation study, thanks to the use of speculative reads, STR can provide up to 6× throughput gains over systems, like PLANET or ICG, that only use speculative commits.

The idea of letting transactions "optimistically" borrow, in a controlled manner, data updated by concurrent transactions has already been investigated in the past. SPECULA [35] and Aggro [32] have applied this idea to local area clusters in which data is fully replicated via total-order based coordination primitives; Jones et. al. [25] applied this idea to partially replicated/distributed databases, by relying on a central coordinator to totally order distributed transactions. These solutions provide consistency guarantees on executing transactions (and not only on committed ones) that are similar in spirit to the ones specified by SPSI[1]. However, these systems rely on solutions (like a centralized transaction coordinator or global sequencer) that impose unacceptably large overheads in geo-distributed settings.

Other works in the distributed database literature, e.g., [22, 34, 18], have explored the idea of speculative reads (sometimes referred to as *early lock release*) in decentralized transactional protocols for partitioned databases, i.e., the same system model assumed by STR. However, these protocols provide no guarantees on the consistency of the snapshots observed by transactions (that eventually abort) during their execution and may expose applications to subtle concurrency bugs such as the ones exemplified in Figure 1.

Mixing consistency levels. Some recent systems exploit the coexistence of multiple consistency levels to enhance system performance. Gemini [29] and Indigo [7] identify and exploit the presence of commutative operations that can be executed with lightweight synchronization schemes, i.e. causal consistency, without breaking application invariants. These techniques are orthogonal to STR, which tackles the problem of enhancing the performance of non-commutative transactions that demand stronger consistency criteria (i.e., SI). Salt [48] introduced the notion of BASE transactions, i.e., a classic ACID transaction that is chopped into a sequence of sub-transactions, which can externalize intermediate states of their encompassing transaction to other BASE transactions. This approach, analogously to STR's speculative reads, allows to reduce lock duration and enhance throughput. Differently from STR, though, Salt requires programmers to define which intermediate states of which BASE transactions should be externalized and to reason on the correctness implications of exposing such states to other BASE transactions. STR's SPSI semantics spare programmers from this source of complexity, by ensuring that transactions always observe and produce atomic and isolated snapshots — which are guaranteed not to include the execution of concurrent transactions originated at different nodes.

[1] In fact, these works do not consider SI as base consistency criterion, but rather opacity [20] and serializability.

3 System and transaction execution model

Our target system model encompasses a set of geo-distributed data centers, each hosting a set of nodes. In the following, we shall assume a key-value data model. This is done for simplicity and since our current implementation of STR runs on a key-value store. However, the protocol we present is agnostic to the underlying data model (e.g., relational or object-oriented).

Data and replication model. The dataset is split into multiple partitions, each of which is responsible for a disjoint key range and maintains multiple timestamped versions for each key. Partitions may be scattered across the nodes in the system using arbitrary data placement policies. Each node may host multiple partitions, but no node or data center is required to host all partitions.

A partition can be replicated within a data center and across data centers. STR employs synchronous master-slave replication to enforce fault tolerance and transparent fail over, as used, e.g., in [12, 6]. A partition has a master replica and several slave replicas. We say that a key/partition is remote for a node, if that node does not replicate that key/partition. At commit time, update transactions contact the masters of the partitions they accessed. These verify whether transactions can be correctly serialized and propagate their updates, along with any metadata (e.g., locks held by the transaction) required for their recovery, to its replicas. This scheme allows for transparent fail over, in master replicas fail. Further, it allows reads to be served by any replica, which allows clients to freely select their geographically closest ones.

Synchrony assumptions. STR does not rely on any synchrony assumption, except that for the management of failures. STR only requires that nodes are equipped with loosely synchronized, conventional hardware clocks, which we only assume to monotonically move forward. Additional synchrony assumptions, though, are required to ensure the correctness of the synchronous master-slave replication scheme, used by STR, in presence of failures [17]. STR integrates a classic single-master replication protocol, which assumes perfect failure detection capabilities [11]. However, it would be relatively straightforward to replace the replication scheme currently employed in STR to use techniques, like Paxos [15], which require weaker synchrony assumptions.

Transaction execution model. Transactions are first executed in the node where they were originated. When they request to commit, they undergo a local certification phase, which checks for conflicts with concurrent transactions, originated either locally or remotely. If the local certification phase succeeds, we say that transactions *local commit* and are attributed a local commit timestamp, noted *LC*. Next, they execute a global certification phase that detects conflicts with transactions originated at any other node in the system. Transactions that pass the global certification phase are said to *final commit* and are attributed a final commit timestamp, noted *FC*.

A local committed transaction, T, can expose its state to other transactions via the speculative read mechanism. We say that these transactions have *data dependencies* on T. Programmers can also allow to expose the state produced by a local committed transaction, T, to clients via the speculative commit mechanism. Then clients can activate new transactions without waiting for the final commit of T. Such transactions are said to *flow depend* on T.

4 Programming Model

As discussed in§1, STR uses both internal (speculative reads) and external (speculative commits) speculation techniques. While the former ones are totally transparent to programmers, speculative commits allow to expose uncommitted state and, as such, require the development of compensation logic to deal with misspeculations. To this end, STR employs an API, similar in spirit to the ones proposed by other recent systems [33, 21], which allows developers to circumscribe the scenarios in which external speculation should be used and to define ad-hoc compensation logics. We exemplify STR's API by means of a simple online shopping application (Listing 1), which allows users to purchase an item and decrements its quantity by one.

```
buyItemTx(String itemKey) {
  CanSpecCommit checkRisk= new CanSpecCommit() {
    public boolean canSpecCommit(TxInfo txInfo)
    {return txInfo.get("itemPrice") < 100
         && SYSINFO.getCommitProb("buyItem") >0.9;} };
  OnSpecCommit ackOrder
       = //Display "Your order has been placed."
  OnFinalCommit confirmOrder = //Send an
       email to the user notifying successful order.
  try {
    Transaction tx = new Transaction();
    Item item = tx.read(itemKey);
    item.quantity -= 1;
    tx.write(itemKey, item);
    tx.getTxInfo().put("itemPrice", item.price);
    tx.commit(checkRisk, ackOrder, confirmOrder);
  }
  catch (NonSpecTxAbortException e1) {
    // Retry.
  }
  catch (SpecTxAbortException e2) {
    // Send apology email to the client.
  }
}
```

Listing 1: Exemplifying STR's programming model.

STR extends the API exposed by conventional, non-speculative transactional systems in a simple and intuitive way, by requiring programmers to specify, upon transaction commit, the following three callbacks:

• CanSpecCommit() is invoked by STR when a transaction completes its local execution, and returns a boolean that determines whether STR should or should not speculative commit the transaction. In the example, this callback (implemented by *checkRisk*()) evaluates the risk associated with external speculation on the basis of the price of the item being sold and on the commit rate experienced by the corresponding transaction over a recent time window. The former information is inserted during transaction execution in the *txInfo* map, which is associated with the specific transaction instance. The statistical information on the commit probability of various transaction types are instead obtained via *SYSINFO*, a shared in-memory map that is maintained by STR.

• onSpecCommit() is invoked if the transaction is allowed to speculative commit (CanSpecCommit() returns true) and allows for defining how the transaction's speculative state should be exposed — in the example, it informs the user that the order has been placed.

• onFinalCommit(), as the name suggests, is invoked if the transaction successfully finalizes its global certification phase, i.e., it final commits — in the example, it confirms the success of the purchase via email.

Finally, STR lets programmers react to the abort of transactions that exposed speculative state via the *SpecTxAbortException* (sending an apology email in the example), as well as of transactions that did not externalize uncommitted states via the *NonSpecTxAbortException* (in which case the transaction can be simply retried).

5 Speculative Snapshot Isolation

SPSI has been designed to generalize the well-known SI criterion and define a set of rigorous, yet intuitive, guarantees that shelter applications from the subtle anomalies (exemplified in Figure 1) that may arise when using speculative techniques. Before presenting the SPSI specification, let us first recall the definition of SI [46]:

- SI-1. (*Snapshot Read*) All operations read the most recent committed version as of the time when the transaction began.
- SI-2. (*No Write-Write Conflicts*) The write-sets of any committed concurrent transaction must be disjoint.

Let us now introduce the SPSI specification:

- SPSI-1. (*Speculative Snapshot Read*) A transaction T originated at a node N at time t must observe the most recent versions created by transactions that i) final commit with timestamp $FC \leq t$ (independently of the node where these transactions originated), or ii) local committed with timestamp $LC \leq t$ and originated at node N.
- SPSI-2. (*No Write-Write Conflicts among Final Committed Transactions*) The write-sets of any final committed concurrent transaction must be disjoint.
- SPSI-3. (*No Write-Write Conflicts among Transactions in a Speculative Snapshot*) Let S be the set of transactions included in a snapshot. The write-sets of any concurrent transaction in S must be disjoint.
- SPSI-4. (*No Dependencies from Uncommitted Transactions*) A transaction can only be final committed if it does not data or flow depend on any local-committed or aborted transaction.

SPSI-1 extends the notion of snapshot, at the basis of the SI definition, to provide the illusion that transactions execute on immutable snapshots, which reflect the execution of all the transactions that local committed before their activation and originated on the same node. By demanding that the snapshots over which transactions execute reflect *only* the effects of locally activated transactions, SPSI allows for efficient implementations, like STR, which can decide whether it is safe to observe the effects of a local committed transaction based solely on local information. Note that this guarantee applies to *every* transaction, including those that are eventually aborted. SPSI-1 has also another relevant implication: assume that a transaction T, which started at time t, reads speculatively from a local committed transaction T' with timestamp $LC \leq t$, and that, later on, T' final commits with timestamp $FC > t$; at this point T violates the first sub-property of SPSI-1. Hence, T must be aborted before T' is allowed to final commit.

SPSI-2 coincides with SI-2, ensuring the absence of write-write conflicts among concurrent final committed transactions. SPSI-3 complements SPSI-1 by ensuring that the effects of conflicting transactions can never be observed. Finally, SPSI-4 ensures that a transaction can be final committed only if it does not depend on transactions that may eventually abort.

Overall, SPSI restricts the spectrum of anomalies that can be experienced by local committed transactions, by limiting them only to conflicts with concurrent transactions originated at remote sites and of which the local node is not aware yet. More formally, SPSI ensures that any transaction T, which uses speculative reads and speculative commits, observes/produces snapshots equivalent to the ones that T would have produced/observed, if it had executed in a SI-compliant history that included only the transactions known by the node in which T originated, at the time in which T was activated.

6 The STR protocol

For sake of clarity, the design of STR is presented in an incremental fashion. We first present a non-speculative protocol, which represents the basis on top of which STR is built. This base protocol is then extended with a set of mechanisms aimed to support speculation in an efficient and safe (i.e., SPSI compliant) way. Next, we explain the STR protocol along with its pseudo-code. Finally, we discuss the fault-tolerance of STR and explain how the self-tuner adjusts the speculation degree.

6.1 Base non-speculative protocol

The base, non-speculative, protocol on top of which we designed STR is a multi-versioned, SI-compliant algorithm that avoids non-scalable solutions, like the use of centralized sequencers [25] or the involvement in a transaction's certification phase of nodes that do not replicate data accessed by that transaction [40, 5]. Conversely, STR's base protocol relies on a fully decentralized concurrency control scheme that is similar in spirit to the one employed by recent, highly scalable systems like Spanner or Clock-SI [13, 12]. In the following, we describe the main phases of STR's base protocol.

Execution. When a transaction is activated, it is attributed a *read snapshot*, noted as *RS*, equal to the physical time of the node in which it was originated. The read snapshot determines which data item versions are visible to the transaction. Upon a read, a transaction T observes the most recent version v having final commit timestamp $v.FC \leq T.RS$. However, if there exists a pre-committed version v' with a timestamp smaller than $T.RS$, then T must wait until the pre-committed version is committed/aborted. In fact, as it will be clearer shortly, the pre-committed version may eventually commit with a timestamp $FC \leq RS$ — in which case T should include it snapshot — or $FC > RS$ — in which case it should not be visible to T.

Note that read requests can be sent to any replica that maintains the requested data item. Also, if a node receives a read request with a read snapshot *RS* higher than its current physical time, the node delays serving the request until its physical clock catches up with *RS*. Instead, writes are always processed locally and are maintained in a transaction's private buffer during the execution phase.

Certification. Read-only transactions can be immediately committed after they complete execution. Update transactions, instead, first check for write-write conflicts with concurrent local transactions. To this end an update transaction T that is being certified must check whether, for any of the data items it updated and that is locally replicated: i) there exist a pre-committed version created by a transaction T' — in which case T must block until the outcome of T' is determined; or, ii) the most recent final committed version has a timestamp larger than the the read snapshot of T — in which case T has to be aborted.

If T passes this local certification stage, it activates a, 2PC-based, global certification phase by sending a pre-commit request to the master replicas of any key it updated and for which the local node is not a master replica. If a master replica detects no conflict, it acquires pre-commit locks, and proposes its current physical time for the pre-commit timestamp.

Replication: If a master replica successfully pre-commits a transaction, it synchronously replicates the pre-commit request to its slave replicas. These, in their turn, send to the transaction coordinator their physical time as proposed pre-commit timestamps.

Commit: After receiving replies from all the replicas of updated partitions, the coordinator calculates the commit timestamp as the maximum of the received pre-commit timestamps. Then it sends a commit message to all the replicas of updated partitions and replies to the client. Upon receiving a commit message, replicas mark the version as committed and release the pre-commit locks.

6.2 Ensuring atomic and isolated speculative snapshots

Let us now extend the base protocol described above to incorporate speculative reads, i.e., reads of pre-committed versions. The example execution in Fig. 1.a, illustrates a possible anomaly that could arise if one adopted a naive protocol that would simply allow to observe any pre-committed version having pre-committed timestamp smaller than the transaction's read snapshot. Since nodes propose pre-commit timestamps in an independent fashion, and depending on whether they receive the pre-commit request, i.e., asynchronously, transactions could observe non-atomic snapshots, violating property SPSI-1. Furthermore, Fig. 1.b illustrates how such a naive protocol may violate property SPSI-3, allowing to include in $T3$'s snapshot versions created by two conflicting transactions.

STR tackles these issues as follows. First, it restricts the use of speculative reads, as mandated by SPSI-1, by allowing to observe only pre-committed versions created by local transactions. To this end, when a transaction local commits, it stores in the local node the (pre-committed) versions of the data items that it updated and that are also replicated by the local node. This is sufficient to rule out the anomalies illustrated in Fig. 1, but it still does not suffice to ensure properties SPSI-1 and SPSI-3. There are, in fact, two other subtle scenarios that have to be taken into account, both involving speculative reads of versions created by local committed transactions that updated some remote key.

The first scenario, illustrated in Fig. 2, is associated with the possibility of including in the same snapshot a local committed transaction, $T1$ — which will eventually abort due to a remote conflict, say with $T2$ — and a remote, final committed transaction, $T3$, that has read from $T2$. Such an execution clearly violates property SPSI-3. It should be noted that the local certification phase, whose success is a necessary condition to local commit a transaction (and, hence, allow exposing its updated versions via speculative reads), can only detect conflicts between local transactions, or between local transactions and remote transactions that pre-commit at the local node. Indeed, the totally decentralized nature of STR's concurrency protocol, in which no node has global knowledge of all the transactions committed in the system, makes it challenging to detect scenarios like the ones illustrated in Figure 2 and to distinguish them, in an exact way, from executions that did not include transaction $T2$ — in which case the inclusion of $T1$ and $T3$ in $T4$ would have been safe.

The mechanism that STR employs to tackle this issue is based on the observation that such scenarios can arise only in case a transaction, like $T4$, attempts to read speculatively from a local committed transaction, like $T1$, which has updated some remote key. The latter type of transactions, which we call "unsafe" transactions, may have in fact developed a remote conflict with some *concurrent* final committed transaction (which may only be detected during their global certification phase), breaking property SPSI-3. In order to detect these

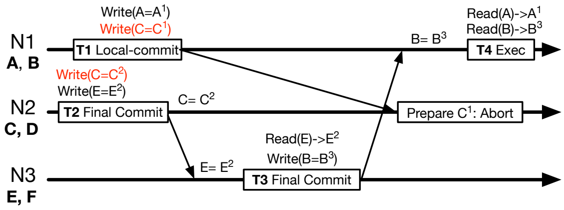


Figure 2: History exemplifying indirect conflicts between a local committed transaction, $T1$, and a final committed transaction originated at a different node, $T3$. If $T4$ included both $T1$ and $T3$ in its snapshot, it would violate SPSI property 3.

scenarios, STR maintains two additional data structures per transaction: *OLC* (Oldest Local-Commit) and *FFC* (Freshest Final Commit), which track, respectively, the read snapshot of the oldest "unsafe" local committed transaction and the commit timestamp of the most recent remote final committed transaction, which the current transaction has read from (either directly or indirectly). Thus, STR blocks transactions when they attempt to read versions that would cause *FFC* to become larger than *OLC*. This mechanism prevents including in the same snapshot of a transaction unsafe local committed transactions along with remote final committed transactions that are concurrent and may conflict with them. For example, in Fig. 2, STR blocks T_4 when attempting to read B from T_3, until the outcome of $T1$ is determined (not shown in the figure).

The second scenario arises in case a transaction T attempts to speculatively read a data item d that was updated by a local committed transaction T', in case d is not replicated locally. In this case, if T attempted to read remotely d, it may risk to miss the version of d created by T', which would violate SPSI-1. To cope with this scenario, whenever an unsafe transaction local commits, it temporarily (until it final commits or aborts) stores the remote keys it updated in a special *cache partition*, tagging them with the same local commit timestamp. This grant prompt and atomic (i.e., all or nothing) access to these keys to any local transaction that may attempt to speculatively read them.

6.3 Maximizing the chances of speculation

Recall that, SPSI-1 requires that if a transaction T reads speculatively from a local committed transaction T', and T' eventually final commits with a commit timestamp that is larger than the read snapshot of T, then T has to be aborted. Thus, in order to increase the chance of success of speculative reads, it is important that the commit timestamps attributed to final committed transactions are "as small as possible".

To this end, STR proposes a new timestamping mechanism, i.e., *PreciseClock*, which is based on the following observation. The smallest final commit timestamp, *FC*, attributable to a transaction T that has read snapshot *RS* must ensure the following properties:

- P1. $T.FC > T.RS$, which is necessary to guarantee that if T reads a data item version with timestamp *RS* and updates it, the version it generates has larger timestamp than the one it read.

- P2. $T.FC$ is larger than the read snapshot of all transactions $T_1,...,T_n$ that read, before T final committed, any of the keys updated by T, and that did not see the versions created by T, i.e., $T.FC > \max\{T_1.RS,...,T_n.RS\}$. This condition is necessary to ensure that T is serialized after the transactions $T_1, \ldots, T_n$, or, in other words, to track write-after-read dependencies among transactions correctly.

Ensuring property P1 is straightforward: instead of proposing the value of the physical clock at its local node as pre-commit timestamp, the transaction coordinator proposes $T.RS+1$. In order to ensure the latter property, STR associates to each data item an additional timestamp, called *LastReader*, which tracks the read snapshot of the most recent transaction to have read that data item. Hence, in order to ensure property P2, it suffices that the nodes involved in the global certification phase of a transaction T propose, as its pre-commit timestamp, the maximum among the *LastReader* timestamps of any key updated by T on that node.

It can be easily seen that the PreciseClock mechanism allows to track write-after-read dependencies among transaction at a finer granularity that the timestamping mechanism used in the base protocol — which, we recall, is also the mechanism used by non-speculative protocols like, e.g., Spanner [12] or Clock-SI [13]. Indeed, as we will show in§ 8, the reduction of commit timestamps, achievable via PreciseClock, does not only increase the chances of successful speculation, but also reduces abort rate for non-speculative protocols.

6.4 Tracking transaction dependencies

SPSI-4 allows transaction to commit only if they have no data or flow dependencies on local committed or aborted transactions. To accomplish this goal, in STR each transaction T maintains the following data structures:

- *inDD*: a set that tracks the transactions which T data depends on. T cannot final commit unless *inDD* is empty.
- *outDD*: a set that tracks the transactions that data depend on T. These transactions are notified when T aborts or final commits. If T aborts, the transactions in *outDD* are aborted as well. If T final commits, T aborts all the transactions in *outDD* that have a read snapshot smaller than T's final commit timestamp (which is needed to ensure SPSI-1, see§ 5), and notifies the remaining transactions to remove T from their *inDD* set.
- *inFD*: it tracks the transaction that T flow depends on, if any. As for *inDD*, T cannot final commit unless *inFD* is empty.
- *outFD*: it tracks the transaction that flow depends on T, if any. T notifies the transaction tracked by *outFD*, say T', when it final commit or aborts. In the former case, T' remove T from its *inDD*. Else, T' aborts.

These data structures allow also for efficiently identifying the cascading abort mechanism: whenever a transaction aborts it just has to notify the transactions in its *outDD* and *outFD* to also abort. Finally, we call the number of pending speculatively-committed transactions that a client has subsequently activated as *speculation chain length*, or, more concisely, SL, which can be tracked by *inFD*.

6.5 Detailed protocol description

In this subsection we provide a detailed description of the STR protocol, whose behavior is formalized by the pseudo-code in Alg. 1 and 2.

Start transaction. A transaction is initialized in a node and assigned a read snapshot (*RS*) equal to the current value of the node's physical clock. It initializes its FFC (to 0) and *OLDDict*, a dictionary that stores *OLC* of transactions it will read from. As the transaction has not read from any unsafe transaction, *OLCDict* is set to contain ∞ (Alg1, 1-6).

Speculative read. Read requests to locally-replicated keys are served by corresponding local partitions directly. A read request to a non-local key is first served at the cache partition to check for updates from previous local-committed transactions. If no appropriate version is found, the request is sent to any (remote) replica of the partition that contains this key (Alg1, 8-12). Upon receiving a read request to a key, a partition updates the *LastReader* of the key and fetches the latest version of the key with a timestamp no larger than the reader's read snapshot (Alg2, 6-7). If the fetched version is committed, or it is local-committed and the reader is reading locally, then the partition returns the value and id of the transaction that created the value; otherwise, the request request is blocked until the transaction's final outcome is known (Alg2, 8-14). Upon receiving the read reply, the reader transaction updates its *OLCDict* and FFC, and only reads the fetched value if the minimal value of its *OLCDict* is greater than or equal to its FFC. If not, this value may conflict with other values already included in the transaction's snapshot, so the transaction waits until the minimal value in its *OLCDict* becomes larger than its FFC (Alg1, 13-15). This condition may never become true if the transaction that created the fetched value actually conflicts with transactions already contained in the reader's snapshot. In that case, the reader will be notified after this conflict is detected and it will abort.

Local certification. After the transaction finishes execution, its write-set is locally certified. The local certification is essentially a local 2PC across all local partitions that contain keys in the transaction's write-set, including the cache partition if the transaction updated non-local keys (Alg1, 18-23). Each partition prepares the transaction if no write-write is detected, and proposes a prepare timestamp according to the PreciseClock rule (Alg2, 16-25). Upon receiving replies from all updated local partitions (including the cache partition), the coordinator calculates the local-commit timestamp as the maximal between received prepare timestamps and the transaction's read snapshot plus one, then notifies all updated local partitions. A notified partition converts the pre-committed record to local committed state with the local commit timestamp (Alg1, 27 and Alg2, 26-30). If the transaction updates non-local keys, the transaction is an 'unsafe' transaction so it adds its own read snapshot to its *OLCDict* (Alg1, 24-25).

After a transaction is local committed, if the provided CANSPECCOMMIT permits, the system executes the SPECCOMMIT callback. Then, the client that issued the transaction may be notified of the "speculative commit" event, and he can issue new transactions until the maximal speculation chain length is reached.

Global certification and replication. After local certification, the transaction performs global certification by sending keys whose master partitions are stored remotely to their corresponding master partitions (Alg1, 28). Just like for the local certification phase, master partitions check for conflicts, propose a prepare timestamp and prepare the transaction (Alg2, 16-22). Then, a master partition replicates the prepare request to its slave replicas and replies to the coordinator (Alg2, 23-25). After receiving a replicated prepare request, a slave partition aborts any conflicting local committed transactions and stores the prepare records. As slave replicas can be directly read bypassing their master replica, slave replicas also track the *LastReader* for keys, so each slave replica also proposes a prepare timestamp for the transaction and replies to the transaction coordinator (Alg2, 32-36).

Final commit/abort. A transaction coordinator can final commit a transaction, if (i) it has received prepare replies from all replicas of updated partitions, (ii) all data dependencies are resolved, and (iii) its flow-dependent transaction has committed. To commit a transaction T, its transaction coordinator first notifies transactions in its *outDD*, i.e., transactions that data-dependent on T: if the read snapshot of a transaction in T's *outDD* is smaller than T's commit timestamp, the transaction is aborted; otherwise, the transaction removes T from its *inDD* and *OLCDict*, and updates its FFC by including T's commit timestamp. Then, the transaction coordinator atomically sets its local committed updates to committed state and cleans its cached updates in the cache partition, if there is any. Then the commit decision, along with the commit timestamp (the maximal of all received prepare timestamps), is send to to all replicas of updated partitions. T's FFC is updated to its own commit timestamp, and its *OLCDict* is set to infinity (Alg1, 40-50).

On the other hand, a transaction is aborted if its certification check fails, its speculative reads are invalid, or its flow-dependent transaction is aborted. The coordinator atomically removes its local-commit updates, aborts transactions flow-dependent or data-dependent on it and sends the decision to remote replicas (Alg1, 52-54).

6.6 Fault tolerance

With respect to conventional/non-speculative 2PC based transactional systems, STR does not introduce additional sources of complexity for the handling of failures.

Algorithm 1: Coordinator protocol

```
1  startTx()
2      Tx.RS←current_time()
3      Tx.Coord←self()
4      Tx.OLCDict←{self(),∞}
5      Tx.FFC←0
6      return Tx

7  read(Tx, Key)
8      if Key is locally replicated or in cache then
9          {Value, Tw}← local_partition(Key).readFrom(Tx, Key)
10     else
11         send {read,Tx,Key} to any p ∈ Key.partitions()
12         wait receive {Value, Tw}
13     Tx.OLCDict.put(Tw, min_value(Tw.OLCDict)}
14     Tx.FFC←max(Tx.FFC, Tw.FFC)
15     return Value when min_value(Tx.OLCDict) >= Tx.FFC

16 commitTx(Tx, canSpecCommit, scCallback, fcCallback)
       // Local certification
17     LCTime←Tx.RS+1
18     for P, Keys ∈ Tx.WriteSet
19         if local_replica(P).prepare(Tx) = {prepared, TS}
20             LCTime← max(LCTime, TS)
21         else
22             abort(Tx)
23             throw NonSpecTxAbortException
24     if Tx updates non-local keys
25         Tx.OLCDict.put(self(), Tx.RS)
26     send local commit to local replicas of updated partitions
27     if canSpecCommit(): scCallback(Tx.getTxInfo())
       // Global certification
28     send prepare to remote masters of updated partitions
29     wait receive {prepared, TS} from Tx.InvolvedReplicas
30         wait until all dependencies are solved
31         CommitTime←max(all received TS)
32         commit(Tx, CommitTime)
33         if Tx.HasSpecCommit: fcCallback()
34         return committed
35     wait receive aborted
36         abort(Tx)
37         if Tx.HasSpecCommit: throw SpecTxAbortException
38         else: throw NonSpecTxAbortException

39 commit(Tx, CT)
40     Tx.FFC←CT
41     Tx.OLCDict←{self(),∞}
42     for Tr with data dependencies from Tx
43         if Tr.RS >= CT then
44             remove Tx from Tr's read dependency
45             Tr.OLCDict.remove(Tx)
46             Tr.FFC←max(Tr.FFC, CT)
47         else
48             abort(Tr)
49     atomically commit Tx's local committed updates
               and remove Tx's cached updates
50     send commit to remote replicas of updated partitions

51 abort(Tx)
52     atomically remove Tx's local committed updates
53     abort transactions with dependencies from Tx
54     send abort to remote replicas of updated partitions
```

Just like any other approach, e.g., [12, 13, 36, 37], based on 2PC, some additional, orthogonal solutions have to be adopted in order to ensure the high availability of the coordinator state. A typical approach, in this sense, consists in replicating the coordinator state, like, e.g., in [19], over a (typically small) set of processes co-located in the same data center, so to minimize the impact of this fault-tolerance technique on the user-perceived performance. These replication techniques can be straightforwardly exploited to ensure the recoverability also of the trans-

Algorithm 2: Partition protocol

```
upon receiving {read, Tx, Key} by partition P
    reply P.readFrom(Tx, Key)

upon receiving {prepare, Tx, Updates} by partition P
    reply P.prepare(Tx, Updates)

readFrom(Tx, Key)
    Key.LastReader←max(Key.LastReader, Tx.RS)
    {Tw, State, Value}←KVStore.latest_before(Key, Tx.RS)
    if State = committed
        return{Value, Tw}
    else if State = local-committed and local_read()
        add data dependence from Tx to Tw
        return{Value, Tw}
    else
        Tw.WaitingReaders.add(Tx)

prepare(Tx, Updates)
    if exists any concurrent conflicting
            local committed or committed transaction
        return aborted
    else
        PT←max(K.LastReader+1 for K ∈ Updates)
        for{K, V} ∈ Updates do
            KVStore.insert(K,{Tx, pre-committed, PT, V})
        if P.isMaster() = true
            send {replicate, Tx} to its replicas
        return {prepared, PrepTime}

localCommit(Tx, LCT, Updates)
    for{K, V} ∈ Updates do
        KVStore.update(K,{Tx, local-committed, LCT, V})
        unblock waiting preparing transactions
        reply to waiting readers

upon receiving {replicate, Tx, Updates}
    abort all conflicting pre-committed transactions
        and transactions read from them
    PT←max(K.LastReader+1 for K ∈ Updates)
    for{K, V} ∈ Updates do
        KVStore.insert(K,{Tx, pre-committed, PT, V})
    reply {prepared, PT}to Tx.Coord
```

actions' results externalized to clients, in case programmers decide to enable the use of STR's external speculation techniques.

As already mentioned in Section 3, currently STR relies on synchronous replication scheme of the pre-commit logs, which requires that the transaction coordinator collects replies from all the nodes that replicate data updated by the transaction. This approach assumes the existence of an underlying group management toolkit providing, e.g., virtual synchrony guarantees [10]. It is then straightforward, upon a view change event, to purge faulty nodes and reconfigure the system to ensure progress. However, replacing the current, synchronous master-slave replication scheme of the pre-commit logs with alternative approaches, based on consensus algorithms like Paxos [28] or Raft [31], would not raise major difficulties.

6.7 Chasing the optimal speculation degree

Both speculative reads and speculative commits are based on the optimistic assumption that local-committed transactions are unlikely to experience contention with remote transactions. Although our experiments in§ 8 show that this assumption is met in well-known benchmarks such as TPC-C and RUBiS, it is highly application-dependent. In fact, the unprejudiced use of speculation in adverse workloads can lead the system to suffer from excessive misspeculation that degrades performance. In order to enhance the performance robustness of STR, we coupled it with a lightweight self-tuning algorithm that dynamically adjusts STR's speculation degree to meet the workload's characteristics. The self-tuning scheme takes a black-box approach that is agnostic of the data store implementation and also totally transparent to application developers. It relies on a hill-climbing search algorithm that tentatively explores the following increasing degrees of speculation, till a local maximum is found: at the lowest extreme, both speculative reads and speculative commits are disabled; then, only speculative reads are enabled; finally, both speculative reads and speculative commits are enabled, and then the maximum speculation chain length at each client is increased from 1 to a given maximal length.

7 Safety and Liveness

In this section, we prove that STR is safe, i.e., it will not violate any SPSI property, and live, i.e., any transaction may be blocked for finite time, but will eventually commit or abort. The following analysis will assume a failure-free scenario. As discussed in§ 6.6, failures can be addressed using orthogonal techniques.

7.1 Safety

SPSI-1: Speculative Snapshot Read Assume there are two transactions $T1$ and $T2$: $T1$ updates a key K, then local commits with $T1.LC$ and final commits with $T1.FC$; $T2$ tries to read K in a partition P and $T2.RS >= T1.FC >= T1.LC$. We consider all possible interleaving of $T1$'s prepare event and $T2$'s read event in P, and prove that in any considered interleaving, either $T2$ will read $T1$'s update to K, or this interleaving violates our assumption.

- *$T2$ reads K before $T1$ pre-commits*: in this case, the *LastReader* of K will be updated to *RS*, and this causes $T1.FC >= T2.RS + 1$. This contracts the assumption $T2.RS >= T1.FC$.
- *$T2$ reads K after $T1$ pre-commits, but before $T1$ local commits*: first, $T2$ will be blocked until $T1$ local commits (if $T1$ and $T2$ were originated at the same node) or final commits ($T1$ and $T2$ were originated at different nodes). In both case, when $T2$ is unblocked, since $T2.RS >= T1.FC >= T1.LC$, $T2$ will read from $T1$'s update to K.
- *$T2$ reads K after $T1$ local commits but before $T1$ final commits*: if $T1$ and $T2$ were originated at the same node, $T2$ reads $T1$'s update on K; otherwise, $T2$ is blocked until $T1$ final commits and then reads $T1$'s update on K.
- *$T2$ reads K after $T1$ final commits*: since $T2.RS >= T1.FC$, obviously $T2$ will read $T1$'s update on K.

SPSI-2: No Write-Write Conflicts among Final Committed Transactions This property is trivially ensured by

STR's 2PC-based certification. Essentially, a transaction can only commit if all its updated partitions have prepared its write-set. A partition only prepares a transaction if it does not detect conflict between this transaction and any committed transaction; after preparing a transaction, a partition keeps the prepare record and if it receives prepare requests from transactions that can potentially conflict with this transaction, the partition delays serving these requests until only the prepared transaction is either committed or aborted (as described in§6.1).

SPSI-3: No Write-Write Conflicts among Transactions in a Speculative Snapshot We prove this property by contradiction, assuming the following is true: a transaction $T0$ is not suspended due to having potential conflict in its snapshot, but it has read from (possibly indirectly) two conflicting transactions $T1$ and $T2$. We say that T has indirectly read from T', if there exists a chain of transaction T,T'',...T' where each transaction (directly) reads from its following transaction. Without loss of generality, we assume $T0$ was originated at node N_{T0}, and $T1$ conflicts with $T2$ in a key K. We consider all possible states of $T1$ and $T2$, after they have been read by $T0$:

- *$T1$ and $T2$ are both committed*: this is not possible, as SPSI-2 guarantees that committed transactions can not have write-write conflict.
- *$T1$ and $T2$ are both local committed*: if $T1$ and $T2$ were originated at different nodes, then at most one can be read by $T0$, which contradicts the assumption that both of them have been read by $T0$. However, if both of them were originated at the same node, then since they have write-write conflict, at least one of them would have been aborted during local certification, so they can not be both local committed.
- *One of them is committed and the other is local committed*: without loss of generality, we assume $T1$ is local committed and $T2$ is committed. Also, $T1$ should be originated at N_{T0} so that it would be readable to $T0$.

  We firstly consider the case that K, the key $T1$ and $T2$ have conflict on, is replicated by N_{T0}. In this case, since N_{T0} replicates K, then before $T2$ commits, $T2$ must have been prepared on N_{T0} (since a master partition synchronously replicates prepare requests to all its replicas). If $T2$ is prepared on N_{T0} before $T1$ prepares, then $T1$ can not be local committed because its conflict with $T2$ would have been detected during local certification. On the other hand, if $T1$ has been local committed before $T2$ is prepared on N_{T0}, then before preparing $T2$, N_{T0} will abort $T1$ (as described in§6.5). Overall, if $T0$ reads from $T1$ first, $T0$ will be aborted before $T2$ commits; if $T0$ reads from $T2$ first, then since $T1$ is aborted before $T2$ commits, $T0$ can not read from $T1$. Therefore, $T1$ and $T2$ can not both be included in $T0$'s snapshot.

  Then we consider the case that K is not replicated by N_{T0}, which leads to the following inequations:
  1) $T0.OLC >= T0.FFC$, since $T0$ is not suspended,
  2) $T1.OLC <= T1.RS$, because $T1$ updated a non-local key,
  3) $T0.FFC >= T2.FC$ ($T2$'s final commit timestamp), because $T0$ has either directly or indirectly read from $T2$, so $T0.FFC$ includes $T2$'s final commit timestamp, and
  4) $T1.OLC >= T0.OLC$, since $T0$ has read from $T1$, $T0.OLC$ includes $T1.OLC$.
  By combining the above inequations, we can conclude that $T1.RS >= T2.FC$, which means that $T1$ and $T2$ are not concurrent and does not conflict. This contradicts the assumption.

SPSI-4: No Dependencies from Uncommitted Transactions As described in§6.4, a transaction keeps the identifiers of its data and flow dependent transactions in *inDD* and *inFD*, respectively. A transaction can not commit before both its *inDD* and *inFD* are empty, and a dependency in both sets is only removed if the dependent transaction commits. Therefore, a committed transaction can not data or flow depend on local committed or aborted transactions.

7.2 Liveness

We give a high-level discussion about the liveness of STR with no presence of failure. We consider all possibilities that may block a transaction, T, during its execution and show that T can be not blocked infinitely.

- *Blocked during reading*: T can be blocked for two cases when trying to read a key: 1) the latest version of the key is pre-committed with a pre-commit timestamp smaller than or equal to T's snapshot time, or 2) the latest version of the key is a local committed version with a local commit timestamp smaller than T's snapshot time, and T is not reading locally. In both cases, T will be unblocked until the pre-committed or local committed transaction gets finalized (committed or aborted).
- *Blocked during certification*: during T's certification phase (either local or global), an involved partition can not immediately decide whether to prepare T if the keys in T's write-set have already been prepared by other transactions. Though, we use wait-die scheme [39] base on transaction id to decide if the transaction should wait or simply abort. Deadlock is not possible and T is guaranteed to only wait for finite time.
- *Blocked due to FFC larger than OLC*: if T's FFC is larger than its *OLC*, T is blocked. On one hand, when an unsafe transaction that T has read from gets aborted, T is notified, then it stops waiting and gets aborted. On the other hand, when an unsafe transaction that T has read from gets committed, its *OLC* is removed from $T.OLCDict$. As more transactions are removed from $T.OLCDict$ and no transaction is added, the minimal value of $T.OLCDict$ will eventually be larger than its FFC and T will continue execution.
- *Blocked due to speculative dependencies*: T may not be able to commit because it still has data/flow, i.e. speculative, dependencies. We represent T's speculative dependency chain as T,T',...T'',T''', where each transaction data/flow

depends on its following transaction and T''' has no speculative dependency. Since T can only data/flow depend on transactions with smaller read snapshots than T, this chain is guaranteed to be acyclic. As T''' has no speculative dependency and our previous proof has shown that even if T''' is blocked, T''' will only be blocked for finite time. As such, each transaction in the chain may only be blocked for finite time, so eventually T's dependencies will be removed and T will commit.

8 Evaluation

This section reports the results of an extensive experimental study aimed to assess the performance of STR when using exclusively internal speculation, i.e., speculative reads, as well as when jointly enabling external speculation, i.e., when externalizing the results produced by speculatively committed transactions to clients. In the following, we refer to the first STR's variant as STR-Internal, and to the second one as STR-External. This choice allows us to contrast the performance achievable by STR when speculation is used in a fully transparent way to clients and programmers, with the case in which applications can tolerate the risk of exposing misspeculations to clients.

Unless otherwise specified, both STR variants use the hill-climbing self-tuning mechanism described in§ 6.7. For the case of STR-Internal, the self-tuning mechanism simply determines whether to use or not speculative reads. With STR-External, we allow all transactions to speculative commit, and the self-tuning mechanism determines both whether to use speculative reads and the maximum length of the speculation chain length at each client in the [0, 8] range.

We use two baseline protocols. The first one is the non-speculative protocol described in§6.1, which we refer to as ClockSI-Rep, since its execution resembles that of ClockSI [13] extended to support replication. The second baseline aims to emulate protocols, like PLANET [33], which allows for speculatively committing transactions to reduce user-perceived latency. Unlike STR, though, PLANET builds on a non-speculative data store, and as such it does not allow speculative reads nor the speculative commit of more than a transaction at a single client. We implement this baseline by building it atop ClockSI-Rep and allowing clients to speculatively commit at most one transaction (SL=1). Note that the original PLANET system relies on analytical model, which is designed to predict the likelihood of successful external speculation under the assumption that the underlying transactional protocol (MDCC) ensures a much weaker consistency criterion (read committed) than the one adopted by STR. As developing a similar analytical model for SPSI (or even SI) is far from being a trivial task, we could not include this component in this baseline.

Experimental setup We implemented the baseline protocols and STR in Erlang, based on *Antidote*[1], an open-source platform for evaluating distributed consistency protocols. The code of the protocols and of the benchmarks used in this study is freely available at this URL [3].

Our experimental testbed is deployed across the following nine DCs of Amazon EC2: Ireland(IE), Seoul(SU), Sydney(SY), Oregon(OR), Singapore(SG), North California(CA), Frankfurt(FR), Tokyo(TY) and North Virginia(VA). Each DC consists of three m4.large instances (2 VCPU and 8 GB of memory). We use a replication factor of six, so each partition has six replicas, and each instance holds one master replica of a partition and slave replicas of five other partitions. The above list of DCs also indicates the order of replication, e.g., a master partition located at IE has its slave replicas in SU, SY, OR, SG and CA.

A workload stressor is located at each node of the system, which spawns one thread per emulated client. Each client issues transactions to a pool of local transaction coordinators and retries a transaction if it gets aborted. We use two metrics to evaluate latency: the *final latency* of a transaction is calculated as the time elapsed since its first activation until its final commit (including possible aborts and retries); the *perceived latency* is defined as the time since the first activation of a transaction until its last speculative commit, i.e., the one after which it is final committed. Each reported result is obtained from the average of at least three runs. As the standard deviations are low, we omit reporting them in the plots to enhance readability.

8.1 Synthetic workloads

Let us first consider a synthetic benchmark, which allows for generating workloads with precisely identifiable and very heterogeneous characteristics. The synthetic benchmark generates transactions with null "think time", i.e., client threads issue a new transaction as soon as the previous one is final committed, for ClockSI-Rep, STR-Internal and PLANET, or speculatively committed, for STR-External. This type of workload is representative of non-interactive applications, e.g., high frequency trading applications.

Transaction and data access A transaction reads 10 keys then updates them. When accessing a data partition, 90% of the accesses goes to a small set of keys in that data partition, which we call a hotspot, and we adjust the size of the hotspot to control contention rate. Each data partition has two million keys, of which one million are only accessible by locally-initiated transactions and the others are only accessible by remote transactions. This allows adjusting in an independent way the likelihood of contention among transactions initiated by the same local node (local contention) and among transactions originated at remote nodes (remote contention).

We consider three workload scenarios, which we obtain by varying the size of the hotspot size in the local and remote data partitions: i) low local and remote contention, ii) high local and low remote contention, and iii) high local and remote contention.

Low local and remote contention. Let us start by consid-

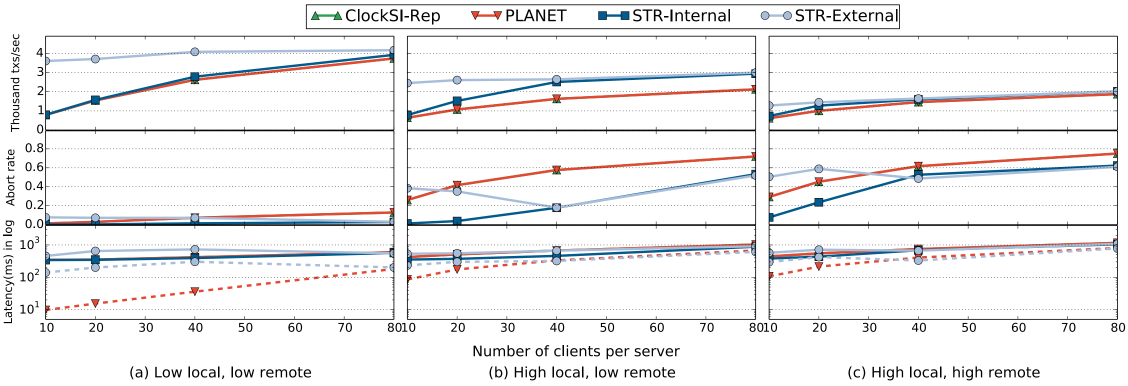


Figure 3: Performance of different protocols under four levels of contention. *Low local, high remote* denotes low local contention and high remote contention, and so forth. In the latency plot, we use solid lines for final latency and dashed lines for perceived latency.

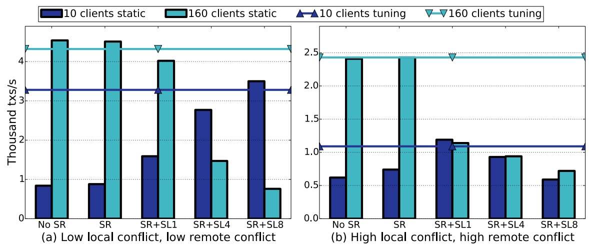


Figure 4: The throughput of tuning versus static configuration. *SR* denotes enabling speculative reads and *SLx* denotes enabling speculative commits with speculation chain length *x*.

ering a workload characterized by low local and remote contention. As shown in Figure 3a, in workloads with negligible contention (both local and remote), STR-Internal, PLANET and ClockSI-Rep (whose throughput/final-latency/abort rate basically overlap with those of PLANET) achieve similar throughput, while STR-External achieves significantly higher throughput up to 40 clients. Intuitively, in STR-Internal, PLANET and ClockSI-Rep clients can only activate new transactions once they have final committed their previous transactions, whereas in the latter, clients can activate new transactions as soon as they have speculatively committed a transaction, unless the maximal speculation chain length is reached. As expectable, the negligible contention of this workload creates also little chance of exploiting the speculative read technique, which explains why STR-Internal and ClockSI-Rep achieve almost identical performance. When the number of clients increases to 80, given the negligible contention level, all protocols fully saturate the available hardware's resources, and reach the peak throughput supported by the system. Still, it is worth highlighting that STR-External is able to saturate the system with a much smaller number of clients than the other protocols.

As we can see from the latency plot, the contrast between the perceived latency of PLANET and STR-External is interesting: while PLANET delivers generally low perceived latency (as there is low contention, transactions rarely abort), the perceived latency of STR-External is higher and varies with the number of clients. This is due to the self-tuning mechanism of STR-External: with small number of clients, the self-tuner tends to choose large speculation chain length to achieve high throughput, which in turn causes the cascading aborts of a large number of transactions when misspeculations occur. However, with larger number of clients, the system becomes increasingly saturated, so the self-tuner chooses smaller speculation chain length. At peak load, i.e., with 80 clients, the self-tuner finally disables pipelining for STR-External, so it provides the same perceived and final latency as PLANET.

Overall, given that with this workload speculative reads are not effective (due to its negligible degree of contention), this experiment allows us also to indirectly quantify the overheads introduced by the concurrency control mechanisms used by STR to support internal speculation (i.e., local certification and speculative reads). Indeed, the fact that the throughputs achieved by ClockSI-Rep, PLANET and STR-Internal in this workload are indistinguishable represents an experimental evidence supporting the efficiency of the proposed mechanisms.

High local and low remote contention. Figure 3b shows a workload with high local and low remote contention. In this workload, due to high local contention, transactions will be frequently blocked if not allowed to read pre-committed data, hence limiting throughput. Also, transactions that pass local certification are likely to commit due to the low remote contention, which means speculative reads can often succeed. Overall, it is a favourable workload for protocols allowing speculative reads, i.e., STR-Internal and STR-External. As Figure 3b shows, both STR-Internal and STR-External achieve much higher throughput than the two baselines even at high number of clients, namely approximately 50% higher throughput at 80 clients. This is due to the fact that the use of speculative reads allows STR to achieve a higher degree of parallelism among transactions, and, hence, a higher peak throughput. Moreover, the abort rate plot shows that the use of PreciseClock greatly reduces abort rate.

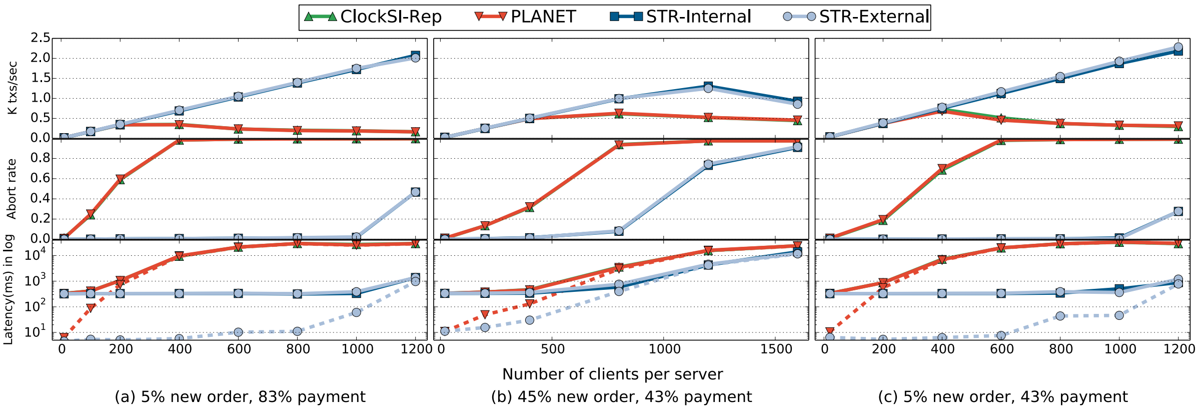


Figure 5: The performance of different protocols for three TPC-C workloads. In the latency plot, we use solid lines for final latency and dashed lines for perceived latency.

However, PLANET achieves no performance gain as transactions often gets blocked during execution and have little chance to speculative commit. The effect of speculative read is directly reflected in latency: when there is low load, the latency of PLANET and ClockSI-Rep increases considerably compared with figure 4a, while the latency of STR-Internal and STR-External are not affected due to speculative read.

High local and remote contention. Lastly, we consider a workload with both high local and remote contention, which is unfavorable for speculative approaches like STR. As Figure 3c shows, all protocols deliver worse throughput than in previous workloads due to high contention. Though, STR still achieves speedup with small number of clients, when the contention is still relatively low. As the number of clients in the systems grows, along with the likelihood of misspeculations, the self-tuning mechanism opts for progressively disabling both speculative reads and pipelining transactions, falling back to a conservative/non-speculative processing mode.

Self-tuning. The previous discussion has shown that STR's self-tuning mechanism allows for delivering robust performance even in adverse workload settings. Figure 4 reports the throughput that STR would achieve using static configurations of the speculation degree. It shows that the speculation degree that maximizes throughput varies significantly, and in non-linear ways, as the workload characteristics vary. The data in Figure 4 does not only highlight the relevance of the self-tuning capabilities of STR, but also provides an experimental evidence of the fact that, once fixed the system's load, the relation between speculation degree and throughput is expressed via convex functions — a necessary condition to ensure convergence to global optimum for local search strategies such as the one employed by STR's self-tuning mechanism. This finding supports the design choice of STR's hill-climbing-based self-tuning strategy, in favour of more complex strategies (like simulated annealing [42]) that sacrifice convergence speed in order to achieve better accuracy in non-convex optimization problems.

# of keys to update	10	20	40	100	200
Speedup	1.02	1.04	1.07	1.18	1.38

Table 1: Speedup of using PreciseClock varying the number of keys to update for each transaction

Coping with fluctuating load levels is also relatively straightforward, as it just requires detecting statistically meaningful changes in the average input load (e.g., by using robust change detectors like the CUSUM algorithm [8]), and react to these events by re-initiating the hill-climbing-based self-tuning mechanism.

Benefits and overhead of PreciseClock. The above experiments have shown that the use of PreciseClock can greatly reduce transaction's abort rate. Intuitively, with lower abort rate, the system wastes less resources aborting and re-executing transactions, which directly benefits throughput. In this experiment, we quantify how the reduction in abort rate due to PreciseClock improves throughput in non-speculative systems. We compare the original ClockSI-Rep with a new version that is equipped with PreciseClock. Our workload varies the number of keys each transaction reads and updates. As shown in Table 1, the more keys a transaction updates, the higher speedup PreciseClock brings. When each transaction accesses 200 keys, PreciseClock improves the throughput by 38%.

We also assessed the additional storage overhead introduced by the use of PreciseClock, which, we recall, requires maintaining additional metadata (a timestamp) for each key. Our measurement shows that for two realistic workloads TPC-C and RUBiS, PreciseClock requires about 9% of extra storage.

8.2 Macro benchmarks

Next, we evaluate the performance of STR with two realistic benchmarks, namely TPC-C[4] and RUBiS [2]. To model realistic human to machine interaction, TPC-C and RUBiS specify large “think time” (instead of null think think for the synthetic ones) between consecutive operations issued by a clients, typically a few seconds. Moreover, some transactions of these two benchmarks, e.g., *payment* of TPC-C and *register_item* of RUBiS, generate much more severe contention levels than the synthetic benchmarks.

TPC-C. We implemented three TPC-C transactions, namely *payment*, *new-order* and *order-status*. The payment transaction has very high local contention and low remote contention; new-order transaction has low local contention and high remote contention, and order-status is a read-only transaction. We consider three workload mixes: 5% new-order, 83% payment and 12% order-status (TPC-C A); 45% new-order, 43% payment and 12% order-status (TPC-C B) and 5% new-order, 43% payment and 52% order-status (TPC-C C). We add the “think time” and “key time” for each transaction as described in the benchmark specification, so a client sleeps for some time (from 10 seconds to as large as hundreds of seconds) both before issuing a new transaction.

Figure 5 shows that speculative reads bring significant throughput gains, as all three workloads have high degree of local contention. Compared with the baseline protocols, STR-Internal and STR-External achieve significant speedup especially for TPC-C A (6.13×), which has the highest degree of local contention due to having large proportion of payment transactions. Though, they still achieve 2.12× and 3× of speedup for TPC-C B and TPC-C C, respectively. Allowing pipelining in this case barely brings speedup, as speculatively-committed transaction usually get final committed before clients “wake up” after a large think time. In terms of latency, though, speculative commits provide significant gains, in terms of reduced perceived latency at the client side: with low number of clients, while the final latency of all protocols is about 400ms, PLANET and STR-External provide about 4ms of perceived latency, an improvement of about 100×.

Another interesting observation is that, with larger number of clients (2000 to 3000), the latency of PLANET and ClockSI-Rep is on the order of 5-8 seconds as a consequence of the high abort rate incurred by these protocols. Conversely, both STR-External and STR-Internal deliver a latency of a few hundred milliseconds.

RUBiS. RUBiS [2] models an online bidding system and encompasses 26 types of transactions, five of which are update transactions. RUBiS is designed to run on top of a SQL database, so we performed the following modifications to adapt it to STR’s key-value store data model: (i) we horizontally partitioned database tables across nodes, so that each node contains an equal portion of data of each table; (ii) we created a local index for each table shard, so that some insertion operations that require a unique ID can obtain the ID locally (instead of modifying a table index shared by all shards by default). We run RUBiS’s 15% update default workload and use its default think time (from 2 to 10 seconds for different transactions).

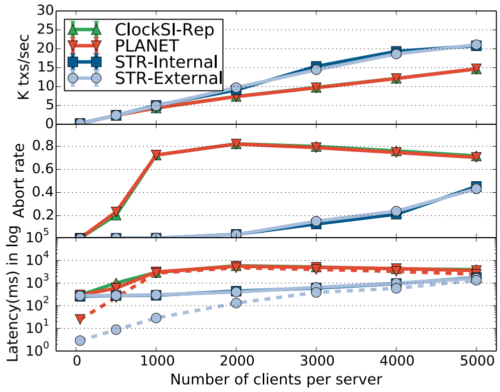


Figure 6: The performance of different protocols for RUBiS. In the latency plot, we use solid lines for final latency and dashed lines for perceived latency.

Also with this benchmark (see Figure 6) STR achieves remarkable throughput gains and latency reduction. With 5000 clients (level at which we hit the memory limit and were unable to load more clients), both STR variants achieve about 43% higher throughput than ClockSI-Rep and PLANET. As for latency, STR-Internal achieves up to 10× latency reduction versus ClockSI-Rep and PLANET, whereas the latency gains extend up to 100× when using STR-External.

9 Conclusion

This paper proposes STR, an innovative protocol that exploits speculative techniques to boost the performance of distributed transactions in geo-replicated settings. STR builds on a novel consistency criterion, which we called SPeculative Snapshot Isolation (SPSI), that extends the familiar SI criterion and shelters programmers from subtle anomalies that can arise when adopting speculative transaction processing techniques. STR combines a set of new speculative techniques with a self-tuning mechanism, achieving striking gains (up to 6× throughput gains and 100× latency reduction) in workloads n workloads characterized by low inter-data center contention, while ensuring robust performance even in adverse settings.

References

[1] antidote. `https://github.com/SyncFree/antidote`.

[2] Rice university bidding system. `http://rubis.ow2.org/`.

[3] Str. `https://github.com/marsleezm/STR`.

[4] Tpc benchmark-w specification v. 1.8. `http://www.tpc.org/tpc_documents_current_versions/pdf/tpc-c_v5.11.0.pdf`.

[5] ARMENDÁRIZ-INIGO, J. E., MAUCH-GOYA, A., DE MENDÍVIL, J., AND MUÑOZ-ESCOÍ, F. D. Sipre: a partial database replication protocol with si replicas. In *Proceedings of the 2008 ACM symposium on Applied computing* (2008), ACM, pp. 2181–2185.

[6] BAKER, J., BOND, C., CORBETT, J. C., FURMAN, J., KHORLIN, A., LARSON, J., LEON, J.-M., LI, Y., LLOYD, A., AND YUSHPRAKH, V. Megastore: Providing scalable, highly available storage for interactive services. In *CIDR* (2011), vol. 11, pp. 223–234.

[7] BALEGAS, V., DUARTE, S., FERREIRA, C., RODRIGUES, R., PREGUIÇA, N., NAJAFZADEH, M., AND SHAPIRO, M. Putting consistency back into eventual consistency. In *Proceedings of the Tenth European Conference on Computer Systems* (2015), ACM, p. 6.

[8] BASSEVILLE, M., AND NIKIFOROV, I. V. *Detection of Abrupt Changes: Theory and Application*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1993.

[9] BERNSTEIN, P. A., HADZILACOS, V., AND GOODMAN, N. Concurrency control and recovery in database systems.

[10] BIRMAN, K., AND JOSEPH, T. *Exploiting virtual synchrony in distributed systems*, vol. 21. ACM, 1987.

[11] CHANDRA, T. D., AND TOUEG, S. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM (JACM) 43*, 2 (1996), 225–267.

[12] CORBETT, J. C., DEAN, J., EPSTEIN, M., FIKES, A., FROST, C., FURMAN, J. J., GHEMAWAT, S., GUBAREV, A., HEISER, C., HOCHSCHILD, P., ET AL. Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems (TOCS) 31*, 3 (2013), 8.

[13] DU, J., ELNIKETY, S., AND ZWAENEPOEL, W. Clock-si: Snapshot isolation for partitioned data stores using loosely synchronized clocks. In *Reliable Distributed Systems (SRDS), 2013 IEEE 32nd International Symposium on* (2013), IEEE, pp. 173–184.

[14] DU, J., SCIASCIA, D., ELNIKETY, S., ZWAENEPOEL, W., AND PEDONE, F. Clock-rsm: Low-latency inter-datacenter state machine replication using loosely synchronized physical clocks. In *Dependable Systems and Networks (DSN), 2014 44th Annual IEEE/IFIP International Conference on* (2014), IEEE, pp. 343–354.

[15] DWORK, C., LYNCH, N., AND STOCKMEYER, L. Consensus in the presence of partial synchrony. *Journal of the ACM (JACM) 35*, 2 (1988), 288–323.

[16] ELNIKETY, S., PEDONE, F., AND ZWAENEPOEL, W. Database replication using generalized snapshot isolation. In *Reliable Distributed Systems, 2005. SRDS 2005. 24th IEEE Symposium on* (2005), IEEE, pp. 73–84.

[17] FISCHER, M. J., LYNCH, N. A., AND PATERSON, M. S. Impossibility of distributed consensus with one faulty process. *Journal of the ACM (JACM) 32*, 2 (1985), 374–382.

[18] GRAEFE, G., LILLIBRIDGE, M., KUNO, H., TUCEK, J., AND VEITCH, A. Controlled lock violation. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data* (2013), ACM, pp. 85–96.

[19] GRAY, J., AND LAMPORT, L. Consensus on transaction commit. *ACM Transactions on Database Systems (TODS) 31*, 1 (2006), 133–160.

[20] GUERRAOUI, R., AND KAPALKA, M. On the correctness of transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (New York, NY, USA, 2008), PPoPP '08, ACM, pp. 175–184.

[21] GUERRAOUI, R., PAVLOVIC, M., AND SEREDINSCHI, D.-A. Incremental consistency guarantees for replicated objects. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)* (GA, 2016), USENIX Association.

[22] HARITSA, J. R., RAMAMRITHAM, K., AND GUPTA, R. The prompt real-time commit protocol. *IEEE Trans. Parallel Distrib. Syst. 11*, 2 (Feb. 2000), 160–181.

[23] HELLAND, P., AND CAMPBELL, D. Building on quicksand. *arXiv preprint arXiv:0909.1788* (2009).

[24] JIMÉNEZ-PERIS, R., PATIÑO MARTÍNEZ, M., KEMME, B., AND ALONSO, G. Improving the scalability of fault-tolerant database clusters. In *Proceedings of the 22 Nd International Conference on Distributed Computing Systems (ICDCS'02)* (Washington, DC, USA, 2002), ICDCS '02, IEEE Computer Society, pp. 477–.

[25] JONES, E. P., ABADI, D. J., AND MADDEN, S. Low overhead concurrency control for partitioned main memory databases. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data* (2010), ACM, pp. 603–614.

[26] KOTLA, R., BALAKRISHNAN, M., TERRY, D., AND AGUILERA, M. K. Transactions with consistency choices on geo-replicated cloud storage. Tech. rep., September 2013.

[27] KRASKA, T., PANG, G., FRANKLIN, M. J., MADDEN, S., AND FEKETE, A. Mdcc: Multi-data center consistency. In *Proceedings of the 8th ACM European Conference on Computer Systems* (2013), ACM, pp. 113–126.

[28] LAMPORT, L. The part-time parliament. *ACM Trans. Comput. Syst. 16*, 2 (May 1998), 133–169.

[29] LI, C., PORTO, D., CLEMENT, A., GEHRKE, J., PREGUIÇA, N. M., AND RODRIGUES, R. Making geo-replicated systems fast as possible, consistent when necessary. In *OSDI* (2012), pp. 265–278.

[30] MAHMOUD, H., NAWAB, F., PUCHER, A., AGRAWAL, D., AND EL ABBADI, A. Low-latency multi-datacenter databases using replicated commit. *Proceedings of the VLDB Endowment 6*, 9 (2013), 661–672.

[31] ONGARO, D., AND OUSTERHOUT, J. K. In search of an understandable consensus algorithm. In *USENIX Annual Technical Conference* (2014), pp. 305–319.

[32] PALMIERI, R., QUAGLIA, F., AND ROMANO, P. Aggro: Boosting stm replication via aggressively optimistic transaction processing. In *Network Computing and Applications (NCA), 2010 9th IEEE International Symposium on* (2010), IEEE, pp. 20–27.

[33] PANG, G., KRASKA, T., FRANKLIN, M. J., AND FEKETE, A. Planet: making progress with commit processing in unpredictable environments. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data* (2014), ACM, pp. 3–14.

[34] PAVLO, A., JONES, E. P., AND ZDONIK, S. On predictive modeling for optimizing transaction execution in parallel oltp systems. *Proceedings of the VLDB Endowment 5*, 2 (2011), 85–96.

[35] PELUSO, S., FERNANDES, J., ROMANO, P., QUAGLIA, F., AND RODRIGUES, L. Specula: Speculative replication of software transactional memory. In *SRDS* (2012), pp. 91–100.

[36] PELUSO, S., ROMANO, P., AND QUAGLIA, F. Score: A scalable one-copy serializable partial replication protocol. In *Proceedings of the 13th International Middleware Conference* (2012), Springer-Verlag New York, Inc., pp. 456–475.

[37] PELUSO, S., RUIVO, P., ROMANO, P., QUAGLIA, F., AND RODRIGUES, L. When scalability meets consistency: Genuine multiversion update-serializable partial data replication. In *Distributed Computing Systems (ICDCS), 2012 IEEE 32nd International Conference on* (2012), IEEE, pp. 455–465.

[38] REN, K., THOMSON, A., AND ABADI, D. J. An evaluation of the advantages and disadvantages of deterministic database systems. PVLDB 7(10): 821-832, 2014.

[39] ROSENKRANTZ, D. J., STEARNS, R. E., AND LEWIS II, P. M. System level concurrency control for distributed database systems. *ACM Transactions on Database Systems (TODS) 3*, 2 (1978), 178–198.

[40] SERRANO, D., PATIÑO-MARTÍNEZ, M., JIMÉNEZ-PERIS, R., AND KEMME, B. Boosting database replication scalability through partial replication and 1-copy-snapshot-isolation. In *Dependable Computing, 2007. PRDC 2007. 13th Pacific Rim International Symposium on* (2007), IEEE, pp. 290–297.

[41] SHUTE, J., VINGRALEK, R., SAMWEL, B., HANDY, B., WHIPKEY, C., ROLLINS, E., OANCEA, M., LITTLEFIELD, K., MENESTRINA, D., ELLNER, S., ET AL. F1: A distributed sql database that scales. *Proceedings of the VLDB Endowment 6*, 11 (2013), 1068–1079.

[42] SUTTON, R. S., AND BARTO, A. G. *Introduction to Reinforcement Learning*, 1st ed. MIT Press, Cambridge, MA, USA, 1998.

[43] TERRY, D. B., THEIMER, M. M., PETERSEN, K., DEMERS, A. J., SPREITZER, M. J., AND HAUSER, C. H. *Managing update conflicts in Bayou, a weakly connected replicated storage system*, vol. 29. ACM, 1995.

[44] THOMSON, A., AND ABADI, D. J. The case for determinism in database systems. *Proceedings of the VLDB Endowment 3*, 1-2 (2010), 70–80.

[45] THOMSON, A., DIAMOND, T., WENG, S.-C., REN, K., SHAO, P., AND ABADI, D. J. Calvin: fast distributed transactions for partitioned database systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data* (2012), ACM, pp. 1–12.

[46] WEIKUM, G., AND VOSSEN, G. *Transactional information systems: theory, algorithms, and the practice of concurrency control and recovery*. Elsevier, 2001.

[47] WOJCIECHOWSKI, P. T., KOBUS, T., AND KOKOCINSKI, M. State-machine and deferred-update replication: Analysis and comparison. *IEEE Transactions on Parallel and Distributed Systems PP*, 99 (2016), 1–1.

[48] XIE, C., SU, C., KAPRITSOS, M., WANG, Y., YAGHMAZADEH, N., ALVISI, L., AND MAHAJAN, P. Salt: Combining acid and base in a distributed database. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)* (2014), pp. 495–509.