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

Abstract

This work presents STR, a geo-distributed, partially replicated transactional data store that leverages speculative techniques to mask the inter-replica synchronization latency. The theoretical foundation of STR is a novel consistency model called 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.

STR provides developers with two levels of speculation: *internal*, which is fully transparent for programmers, as it does not expose the effects of misspeculations to clients; *external*, which does and consequently requires additional compensation logic.

We assess STR's performance on up to nine geo-distributed Amazon EC2 data centers, using both synthetic benchmarks as well as realistic benchmarks (TPC-C and RUBiS). Our evaluation shows that by using only internal speculation STR achieves throughput gains up to 6× and latency reduction up to 10×, in workloads characterized by low inter-data center contention. This is achieved with no additional complexity for the developers. Using external speculation STR can achieve further performance gains, for a total user-perceived latency reduction of up to 100×. Furthermore, thanks to a self-tuning mechanism that dynamically and transparently adjusts the speculation degree, STR offers robust performance even when faced with unfavourable workloads that suffer from high misspeculation rates.

1 Introduction

Modern online services are increasingly deployed over geographically-scattered data centers (geo-replication) [10, 26, 28]. 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 application development [38], some form of global (i.e., inter-data center) certification is needed to safely detect conflicts between concurrent transactions executing at different data centers. The adverse performance impact of global certification is twofold: (i) system 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; and (ii) client-perceived latency is increased, since global certification lies in the critical path of transaction execution.

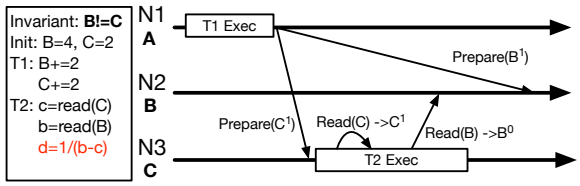
Internal and external speculation. This work investigates the use of speculative processing techniques to alleviate both of the above problems. We focus on geo-distributed partially replicated transactional data stores that provide Snapshot Isolation, a widely employed consistency criterion [11, 14] (SI), and propose a novel distributed concurrency control scheme that employs two key 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 or 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.

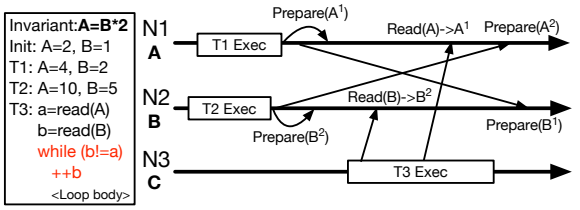
Speculative commits, instead, remove the global certification phase from the critical path of transaction execution. As a consequence, speculative commits can drastically reduce the user-perceived latency, but also expose to external clients the results produced by transactions that are still undergoing their global certification phase. Thus, analogously to other techniques [32, 19] 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.

Avoiding the pitfalls of speculation. Past work has demonstrated how the use of speculative reads and speculative commits, either individually [16, 32, 23, 19] or in synergy [40], can significantly enhance the performance of distributed [40, 34, 23, 32, 33] and single-site [16] transactional systems. However, these approaches suffer from several limitations:

1. **Unfit for geo-distribution/partial replication.** Some



(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.

existing works in this area [34, 23, 40] were not designed for partially replicated geo-replicated data stores. On the contrary, they target different data models (i.e., full replication [34, 40]) or rely on techniques that impose prohibitive costs in WAN environments, such as the use of centralized sequencers to totally order transactions [23].

2. Subtle concurrency anomalies. Existing partially replicated geo-distributed transactional data stores that allow speculative reads [16, 20, 33] 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 with these systems. The root cause of the problem is that existing systems allow speculative reads to observe *any* pre-committed data version. This exposes applications to data snapshots that reflect only partially the updates of transactions (Fig. 1a) and/or include versions created by conflicting concurrent transactions (Fig. 1b). These anomalies have the following negative impacts: (i) transaction execution may be affected to the extent to generate anomalous/unexpected behaviours (e.g., crashing the application or hanging it in infinite loops); and (ii) they can externalize non-atomic/non-isolated snapshots to clients.

3. Performance robustness. If used injudiciously, speculation can hamper 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.

Contributions. This paper presents the following contributions:

- Speculative Transaction Replication (STR), a novel speculative transactional protocol for partially replicated geo-distributed data stores (§6). STR's shares several key design choices with state-of-the-art strongly consistent data stores [10, 11, 35], which contribute to its efficiency and scalability. These include: multi-versioning, which maximizes efficiency in read-dominated workloads [8], purely decentralized concurrency control based on distributed clocks [10, 11, 36], and support for partial replication [24, 10]. The key contribution of STR is its innovative distributed concurrency control scheme that supports speculative execution, while providing intuitive and stringent consistency guarantees.
- Speculative Snapshot Isolation (SPSI), a novel consistency model that is the foundation of STR (§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 read data item versions committed before it started (as in SI), and also to atomically observe the effects of non-conflicting transactions that originated on the same node and pre-committed before it started. This shelters programmers from having to reason about complex concurrency anomalies that can otherwise arise in speculative systems.
- 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 (§6.6).
- We evaluate STR on up to nine geo-distributed Amazon EC2 data centers, using both synthetic and realistic benchmarks (TPC-C [4] and RUBiS [2]) (§7). Our experimental study shows that the use of internal speculation (speculative reads) yields up to 6× throughput improvements and 10× latency reduction in a fully transparent way, i.e., requiring no compensation logic. Further, applications that exploit external speculation (speculative commits) can achieve a reduction of the user-perceived latency by up to 100× and, by allowing clients to overlap the execution of multiple speculative transactions, to boost throughput by up to 4.7×.

2 Related Work

Geo-replication. The problem of designing efficient mechanisms to ensure strong consistency semantics in geo-replicated data stores has been extensively studied. A class of geo-replicated systems [42, 12] is based on the *state-machine replication* (SMR) [27] approach, in which replicas first agree on the serialization order of transactions and then execute them without further coordination. Other recent systems [10, 11, 26, 30] adopt the *deferred update* (DU) [22] approach, in which transactions are first locally executed and then globally certified. This approach is more scalable than SMR in update intensive workloads [44, 22] and, unlike SMR, it can seamlessly support non-deterministic transactions [37]. The main 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 [41]. STR builds on the DU approach and tackles its performance limitation via speculative techniques.

Speculation. To mask latency in replicated systems, Helland et. al. advocated the *guesses and apologies* programming paradigm [21], 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 [32] and ICG [19]. 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 reduce user-perceived latency, but they do not tackle the problem of reducing transaction blocking time, which can severely impair throughput.

The idea of letting transactions "optimistically" borrow, in a controlled manner, data updated by concurrent transactions has already been investigated in the past. SPECULA [34] and Aggro [31] have applied this idea to local area clusters in which data is fully replicated via total-order based coordination primitives; Jones et. al. [23] 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., [20, 33, 16], 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 [28] and Indigo [6] 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 [45] introduced the notion of BASE transactions, i.e., classic ACID transactions that are chopped into a sequence of sub-transactions that can externalize intermediate states of their encompassing transaction to other BASE transactions. This approach, analogously to STR's speculative reads, aims 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.

[1] In fact, these works do not consider SI as base consistency criterion, but rather opacity [18] and serializability.

3 System and data model

Our target system model consists of a set of geo-distributed data centers, each hosting a set of nodes. In the following, we 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 [10, 5]. 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.

Synchrony assumptions. STR requires that nodes be equipped with loosely synchronized, conventional hardware clocks, which we only assume to monotonically move forward. Additional synchrony assumptions are required to ensure the correctness of the synchronous master-slave replication scheme used by STR in presence of failures [15]. STR integrates a classic single-master replication protocol, which assumes perfect failure detection capabilities [9]. However, it would be relatively straightforward to replace the replication scheme currently employed in STR to use techniques, like Paxos [13], 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 in the local node. 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.

Speculation and transaction dependencies. A local committed transaction, T, can expose its state to other transactions via the *speculative read* mechanism. We say that these transactions *data depend* on T. Programmers can also 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 are transparent to programmers, the latter allow to externalize uncommitted state and, as such, require compensation logic to deal with misspeculations. To this end, STR employs an API similar in spirit to the ones proposed by other recent systems [32, 19], which allows developers to use external speculation and to define ad-hoc compensation logic. We exemplify the 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(): 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 is instead obtained via *SYSINFO*, a shared in-memory map that is maintained by STR.

• ONSPECCOMMIT(): invoked if the transaction is allowed to speculative commit (CANSPECCOMMIT() returned 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(): 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's API 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 generalizes the well-known SI criterion and defines a set of guarantees that shelter applications from the subtle anomalies (see Fig. 1) that may arise when using speculative techniques. Before presenting the SPSI specification, we first recall the definition of SI [43]:

- 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 transactions must be disjoint.

We 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 transactions 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 transactions 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's, 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 to conflict 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, we present the design of STR incrementally. We first present a non-speculative base protocol. This base protocol is then extended with a set of mechanisms aimed to support speculation in an efficient and SPSI-compliant way. Finally, we discuss the fault tolerance of STR.

Due to space constraints, we omit the presentation of the data structures used to track transaction flow and data dependencies. We also omit the correctness proof. Both can be found in our extended technical report [29].

6.1 Base non-speculative protocol

The base protocol is a multi-versioned, SI-compliant algorithm that relies on a fully decentralized concurrency control scheme similar to that employed by recent, highly scalable systems, like Spanner or Clock-SI [11, 10]. 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 will become clear shortly, the pre-committed version may eventually commit with a timestamp $FC \leq RS$ — in which case T should include it in 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. 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 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 Enabling SPSI-safe speculations

Let us now extend the base protocol described above to incorporate speculative reads, i.e., reads of pre-committed versions. The example executions in Fig. 1 illustrate two possible anomalies that could lead transactions to observe non-atomic snapshots, which violate property SPSI-1 (Fig. 1.a), or snapshots reflecting the execution of two conflicting transactions, which violate property SPSI-3 (Fig. 1.b).

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 $T2$. In fact, 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 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 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', where 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 grants 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".

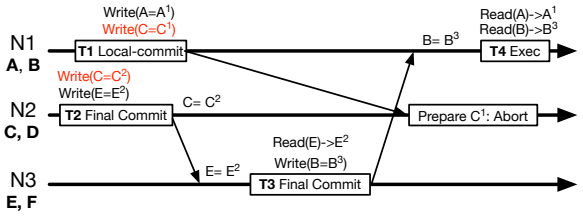


Figure 2: History exemplifying indirect conflicts between a local committed transaction, $T1$, and a final committed transaction originated at a different node, $T3$. If $T4$ included both $T1$ and $T3$ in its snapshot, it would violate SPSI property 3.

To this end, STR proposes a new timestamping mechanism, i.e., *Precise Clock*, 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 guarantees that if T reads a data item version with timestamp RS and updates it, the versions it generates has larger timestamp than the one it read.
- **P2.** $T.FC$ is larger than the read snapshot of all the transactions $T_1,...,T_n$, which (a) read, before T final committed, any of the keys updated by T, and (b) 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 $T_1,...,T_n$, 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 that have read that data item. Hence, in order to ensure property P2, the nodes involved in the global certification phase of transaction T propose, as 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 Precise Clock 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 [10] or Clock-SI [11]. Indeed, as we will show in §7, the reduction of commit timestamps, achievable via Precise Clock, does not only increase the chances of successful speculation, but also reduces abort rate for non-speculative protocols.

6.4 Pseudocode

Algorithms 1 and 2 give the pseudocode of the STR protocol.

Start transaction. Upon activation, a transaction is assigned a read snapshot (RS) equal to the current value of the node's physical clock. Its FFC is set to 0 and its OLCSet, i.e., the set storing

Algorithm 1: Coordinator protocol

```
1  startTx()
2      Tx.RS←current_time()
3      Tx.Coord←self()
4      Tx.OLCSet← {⊥,∞}
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.OLCSet.put(Tw, min_value(Tw.OLCSet)}
14     Tx.FFC←max(Tx.FFC, Tw.FFC)
15     return Value when min_value(Tx.OLCSet) >= 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.OLCSet.put(self(), Tx.RS)
26     send local commit requests to local replicas of updated partitions
27     if canSpecCommit(): scCallback(Tx.getTxInfo())
       // Global certification
28     send prepare requests to remote master of updated partitions
29     wait receive {prepared, TS} from Tx.InvolvedReplicas
30         wait until all dependencies are resolved
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.OLCSet← {⊥,∞}
42     for Tr with data dependencies from Tx
43         if Tr.RS >= CT then
44             remove Tx from Tr's read dependency
45             Tr.OLCSet.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 requests 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 requests to remote replicas of updated partitions
```

the identifiers and read timestamps of the unsafe transactions from which the transaction reads from, to $<\perp,\infty>$ (Alg1, 1-6).

Speculative read. Read requests to locally-replicated keys are served by local partitions. 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 a read request for 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 reader is blocked until the transaction's final outcome is known (Alg2, 8-13). The reader transaction updates its OLCSet and FFC, and only reads the value if the minimum value in its OLCSet is greater than or equal than its FFC. If not, the transaction waits until the minimum value in its OLCSet becomes larger than its FFC (Alg1, 13-15). This condition may never become true if the transaction that created the fetched value conflicts with transactions already contained in the reader's snapshot. In that case, the reader will be aborted after this conflict is detected and stop waiting.

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 Precise Clock rule (Alg2, 15-24). Upon receiving replies from all updated local partitions (including the cache partition), the coordinator calculates the local-commit timestamp as the maximum between the received prepare timestamps and the transaction's read snapshot plus one. Then, it notifies all the updated local partitions. A notified partition converts the precommitted record to local-committed state with the local commit timestamp (Alg1, 27 and Alg2, 25-29). If the transaction updates non-local keys, the transaction is an 'unsafe' transaction, so it adds its snapshot time to its OLCSet (Alg1, 24-25).

After a transaction is local committed, if the CANSPECCOMMIT() fallback permits, the SPECCOMMIT callback is executed. After being notified of the transaction's speculative commit event, clients can issue new transactions until the maximum speculation chain length is reached.

Global certification and replication. After local certification, the keys in the transaction's write-set that have a remote master are sent to their corresponding master partitions for certification (Alg1, 28). As for the local certification phase, master partitions check for conflicts, propose a prepare timestamp and pre-commit the transaction (Alg2, 15-21). Then, a master partition replicates the prepare request to its slave replicas and replies to the coordinator (Alg2, 22-24). After receiving a replicated prepare request, the 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 also proposes a prepare timestamp for the transaction to the coordinator (Alg2, 31-35).

Final commit/abort. A transaction coordinator can final commit a transaction, if (i) it has received prepare replies from all replicas of updated partitions, and (ii) all data dependencies and flow dependencies are resolved. The commit decision, along

with the commit timestamp, is sent to to all non-local replicas of updated partitions. T's FFC is updated to its own commit timestamp, and its $OLCSet$ is set to infinity (Alg1, 40-50). Upon abort, the coordinator removes any local-committed updated version, triggers the abort of any dependent transaction and sends the decision to remote replicas (Alg1, 52-54).

6.5 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. Like any other approach, e.g., [10, 11, 35, 36], based on 2PC, some orthogonal mechanism (typically based on replication [17]) has to be adopted to ensure the high availability of the coordinator state. This mechanism can be straightforwardly exploited to ensure the recoverability also of the transactions' results externalized to clients upon the speculative commit of a transaction.

6.6 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 §7 show that this assumption is met in well-known benchmarks such as TPC-C and RUBiS, this is an application-dependent property. In fact, the unrestrained use of speculation in adverse workloads can lead to excessive misspeculation and degrade performance.

In order to enhance the performance robustness of STR, we coupled it with a lightweight self-tuning mechanism 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, steered by a centralized process that gathers throughput measurements from all nodes in a periodic fashion and tentatively explores increasingly speculative configurations. The search space encompasses the following increasing degrees of speculation: 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 number of speculatively committed transactions at each client, which we call *speculation chain length*, is increased from 1 to a given maximal length. The algorithm stabilizes when it detects to have identified a local maximum.

We opted for a simple and quickly converging search heuristic based on hill-climbing, instead of more complex approaches (e.g., based on off-line trained function approximators or more sophisticated on-line search strategies [39]), since our experimental findings confirm that, for a given workload, the relation between speculation degree and throughput is normally convex (at least this was the case for all the workloads we experimented with) — which is a sufficient condition to guarantee convergence to optimum of local search algorithms like hill-climbing. We provide experimental evidence of this finding in §7.

Algorithm 2: Partition protocol

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

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

5  readFrom(Tx, Key)
6      Key.LastReader←max(Key.LastReader, Tx.RS)
7      {Tw, State, Value}←KVStore.latest_before(Key, Tx.RS)
8      if State = committed
9          return{Value, Tw}
10     else if State = local-committed and local_read()
11         add data dependence from Tx to Tw
12         return{Value, Tw}
13     else
14         Tw.WaitingReaders.add(Tx)

15 prepare(Tx, Updates)
16     if exists any concurrent conflicting transaction
17         return aborted
18     else
19         PT←max(K.LastReader+1 for K ∈ Updates)
20         for{K, V} ∈ Updates do
21             KVStore.insert(K,{Tx, pre-committed, PT, V})
22         if P.isMaster() = true
23             send {replicate, Tx} to its replicas
24         return {prepared, PrepTime}

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

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

7 Evaluation

This section presents an extensive experimental study aimed at answering the following key questions:

1. What performance gains can be achieved by STR if only internal speculation techniques (i.e., speculative reads) are adopted? What performance does instead STR deliver if applications can tolerate the risk of exposing misspeculations to clients, hence allowing STR to combine the use of speculative reads and speculative commits?
2. Which workloads characteristics have the strongest impact on the performance of STR?
3. How relevant is the Precise Clock technique, when used in conjunction with both speculative and non-speculative protocols?
4. How effective is STR's self-tuning mechanism to ensure robust performance in presence of workloads that are not favourable to speculative techniques?

STR variants and baselines. In order to address the first of the above questions, we considered two variants of STR: STR-Internal and STR-External. STR-Internal exclusively uses internal speculation, i.e., speculative reads, while STR-External combines the use of speculative reads and speculative commits, i.e., it can externalize the results produced by speculatively committed transactions to clients.

Unless otherwise specified, both STR variants use the hill-climbing self-tuning mechanism described in §6.6. For the case of STR-Internal, the self-tuning mechanism simply determines whether or not to use speculative reads. With STR-External, we assume programmers allowed all transactions to speculatively commit. So, 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. The self-tuning process gathers throughput measurements with a 10 seconds frequency, and, in all the considered workloads, it stabilizes after at most 3 minutes. The reported results for the STR variants refer to the final configuration identified by the self-tuning process.

We consider 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 [11] extended to support replication. The second baseline aims to emulate protocols, like PLANET [32], 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 to speculatively commit more than a transaction at a single client (i.e., clients cannot submit new transactions before having been notified of the final outcome of a previously speculatively committed transaction), nor the use of speculative reads. We implement this baseline by building it atop ClockSI-Rep and allowing clients to speculatively commit at most one transaction. Note that the original PLANET system relies on an analytical model that 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 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.

Load is injected by spawning one thread per emulated client in some node of the system. 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. Besides reporting abort rate, for PLANET and STR-External we also report their rate of *external misspeculation*, i.e., the percentage of transactions that were speculatively committed but finally aborted. 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.

7.1 Synthetic workloads

Let us start by considering a synthetic benchmark, which allows for generating workloads with precisely identifiable and very heterogeneous characteristics. The synthetic benchmark generates transactions with zero "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. We will consider benchmarks representative of interactive applications in§ 7.2.

Transaction and data access. A transaction reads and updates 10 keys. 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 workloads, which we obtain by varying the size of the hotspot size in the local and remote data partitions: i) high local and low remote contention, ii) low local and remote contention, and iii) high local and high remote contention.

High local and low remote contention. In this workload (Fig. 3a), due to high local contention, transactions will be frequently blocked unless they are allowed to read pre-committed data. Also, given the low remote contention, transactions that pass local certification are likely to commit, which means that speculative reads are likely to succeed. Overall, this is a favourable workload for the use of speculative reads, and this reflects into significant throughput gains for both

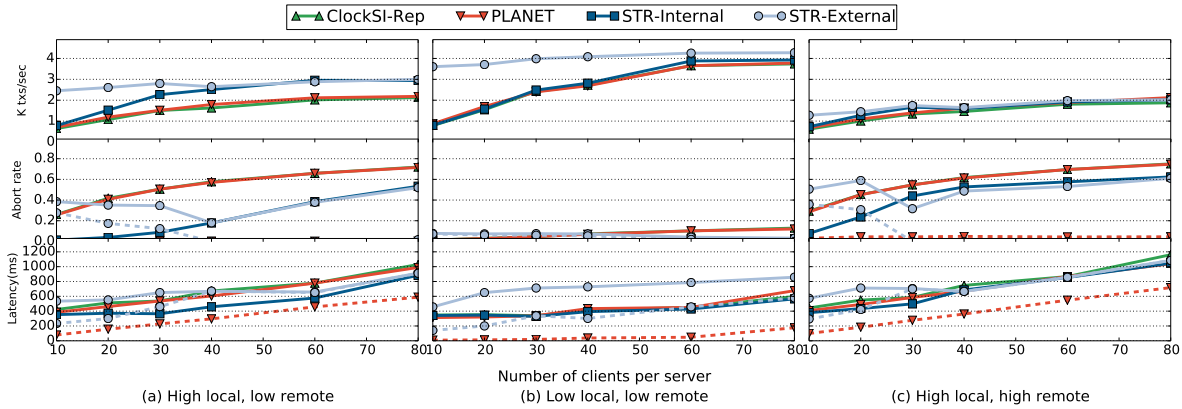


Figure 3: Performance of different protocols under three levels of contention. *Low local, high remote* denotes low local contention and high remote contention, and so forth. In the latency plot, we use solid lines for final latency and dashed lines for perceived latency; in the abort rate plot, we report total abort rate with solid lines and external misspeculation rate with dashed lines.

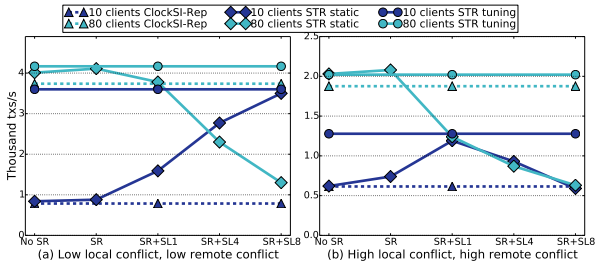


Figure 4: Throughput using self-tuning versus static configurations. *SR* denotes enabling speculative reads; *SLx* denotes enabling speculative commits with speculation chain length *x*.

STR-Internal and STR-External (approx. 50% at 80 clients).

We also note that, at low client count, STR-External outperforms STR-Internal, extending its throughput gains vs the baseline by up to 2.5×. This is due to the fact that, up to 30 clients, the use of speculative commits allows, most of the times, client to successfully "pipeline" new transactions, while still waiting for the outcome of previously submitted ones. As the clients' count grows, however, the contention level grows accordingly, so the chances of successfully pipelining transactions via speculative commits diminish. This is detected by the self-tuning mechanism, which disables the use of external speculation when using more than 30 clients (in fact, the external misspeculation rate drops to 0 above 30 clients for STR-External); this explains why STR-Internal and STR-External yield very similar performance with 60 or more clients.

As for latency, STR-Internal delivers the lowest final latency. In fact, the effective reduction of the lock duration, w.r.t. Clock-SI and PLANET, that is achieved using speculative reads has a beneficial impact both on throughput and latency. STR-External incurs higher final latency than STR-Internal, up to 40 clients. This is due to the fact that, thanks to the use of speculative commits, STR-External achieves much higher throughput levels and, as such, achieves a much higher CPU utilization of the system's nodes. This increase of the system load causes, in its turn, a slow-down of the entire node, and, as a consequence, an increase of the execution time of local processing activities and, ultimately, of the final latency. The same reason explains also why STR-External yields a relatively higher perceived latency w.r.t. PLANET.

Low local and remote contention. This workload (Fig.3b) is expected to be favourable to the use of speculative commit technique, which is used, although in different ways, by both STR-External and PLANET. The low probability of contention reflects, in fact, in a high probability of success of external speculation. Conversely, the lack of contention does not create favourable conditions for the use of speculative reads — as very rarely transactions will attempt to read pre-committed data.

Looking at the throughput plot we observe that 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 (4.7× with 10 clients) up to 40 clients. This can be explained by recalling that STR-External is the only protocol that allows clients to pipeline the execution of transactions, whose processing can be effectively overlapped as long as there are available hardware resources. In other words, STR-External can fully utilize (and saturate) the available hardware resources with a much smaller number of clients. However, when the client count grows to 60 or more, all protocols fully saturate the available hardware's resources (given the low level of data contention), and reach the peak throughput supported by the system.

Analogously to what we observed in the previous workload, STR-External incurs higher latencies, especially at low client count, when compared to the other protocols. Again, this is a consequence of the higher throughput that this protocol sustains, which reflects into a higher load for the system.

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

# of keys / Techniques	10	20	40	100
Physical	1/59%	1/60%	1/60%	1/72%
Precise	1.07/38%	1.07/38%	1.1/35%	1.41/48%
Physical SR	0.68/84%	0.57/83%	0.59/77%	0.97/75%
Precise SR	1.22/47%	1.21/44%	1.31/36%	1.59/49%

Table 1: Normalized throughput/abort rate of different techniques, varying a transaction's number of keys to update. *Physical/Precise* denotes the use of Physical Clock/Precise Clock; *SR* denotes that speculative reads are enabled. Throughputs reported in each column are normalized according to the throughput of 'Physical' in that column.

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 [39]) that sacrifice convergence speed in order to achieve better accuracy in non-convex optimization problems. Moreover, Figure 4 shows that without enabling speculative techniques, STR achieves similar throughput as the non-speculative baseline. This represents an experimental evidence supporting the efficiency of the proposed mechanisms.

Coping with fluctuating load levels can be achieved by detecting statistically meaningful changes in the average input load (e.g., by using robust change detectors like the CUSUM algorithm [7]), and react to these events by re-initiating the hill-climbing-based self-tuning mechanism.

Benefits and overhead of Precise Clock. This experiment aims at quantifying the benefits stemming from the use of the Precise Clock mechanism, when used in conjunction with both speculative and non-speculative protocols. To this end, in Table 1, we consider four alternative systems obtained by considering ClockSI-REP (noted *Physical*) and extending it to use Precise Clocks (noted *Precise*) and/or speculative reads (noted *SR*). In this study we vary the transactions' duration, and hence the corresponding abort cost, by varying the number of keys updated by a transaction. To maintain the contention level stable when increasing the number of keys accessed by transactions, the key space is increased by the same factor.

Table 1 shows that Precise Clock significantly reduces abort rate and can achieve as much as 38% of throughput gain over Physical Clock for a non-speculative protocol. Generally, the more keys transactions update, the larger is the abort cost and the larger the throughput gain achieved by Precise Clock. Another interesting result is that enabling speculative reads with Physical Clock actually has negative effects on abort rate and throughput. In fact, as we have discussed in 6.3, Physical Clock generates large commit timestamp and, thus, rarely allows speculative reads to succeed. Finally, the collective use of Precise Clock and speculative reads results in the best throughput gain (59% for transactions updating 100 keys).

We also assessed the additional storage overhead introduced by the use of Precise Clock, which, we recall, requires maintaining additional metadata (a timestamp) for each accessed key. Our measurement shows that for the TPC-C and RUBiS benchmarks (§7.2), Precise Clock requires about 9% of extra storage.

7.2 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 several seconds of "think time" (instead of zero think time for the synthetic ones) between consecutive operations issued by a client, which essentially eliminates the chance of pipelining transactions to improve throughput.

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) 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 transaction. For TPC-C B and TPC-C C, STR achieve 2.12× and 3× of speedup respectively. Allowing pipelining (STR-External) in this case barely brings speedup, as speculatively-committed transactions usually get final committed before clients issue new transaction after think time.

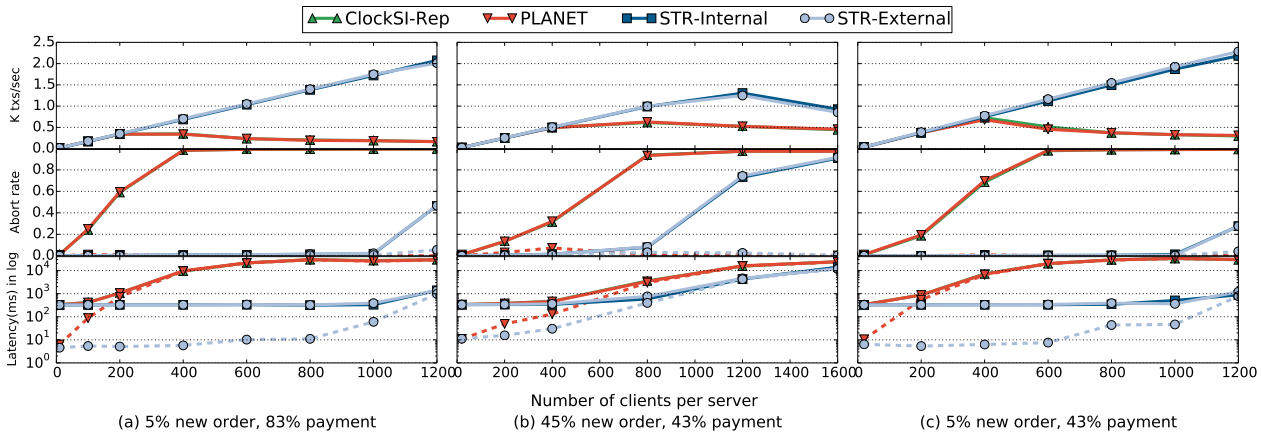


Figure 5: The performance of different protocols for three TPC-C workloads. In the latency plot, we use solid lines for final latency and dashed lines for perceived latency; in the abort rate plot, we report total abort rate with solid lines and misspeculation rate with dashed lines.

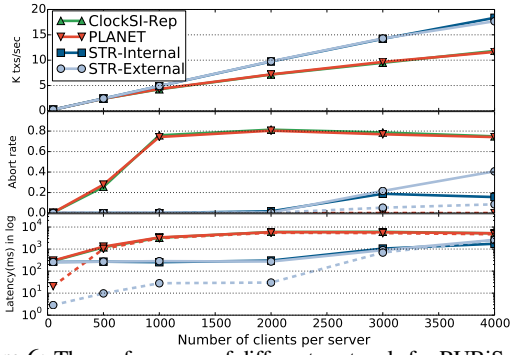


Figure 6: The performance of different protocols for RUBiS. In the latency plot, we use solid lines for final latency and dashed lines for perceived latency; in the abort rate plot, we report total abort rate with solid lines and misspeculation rate with dashed lines.

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×.

Note also 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 still 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 updating 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).

Also with this benchmark (see Figure 6) STR achieves remarkable throughput gains and latency reduction. With 4000 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.

8 Conclusion and future work

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). SPSI 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 increase and 100× latency reduction) in workloads characterized by low inter-data center contention, while ensuring robust performance even in adverse settings.

This work allowed us to identify two main avenues for future research. The first research direction opened by this work is how to adapt both the STR protocol and its underlying speculative correctness criterion to cope with alternative consistency semantics, like Serializability or Strict Serializability. Another interesting research opportunity raised by this work is related to the design and evaluation of alternative self-tuning mechanisms, e.g., based on different modelling methodologies (e.g., relying on white-box analytical models), aimed at optimizing multiple KPIs (e.g., external mispeculation and throughput) or supporting diverse speculation degrees across different nodes.

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] J. Baker, C. Bond, J. C. Corbett, J. Furman, A. Khorlin, J. Larson, J.-M. Leon, Y. Li, A. Lloyd, and V. Yushprakh. Megastore: Providing scalable, highly available storage for interactive services. In *CIDR*, volume 11, pages 223–234, 2011.

[6] V. Balegas, S. Duarte, C. Ferreira, R. Rodrigues, N. Preguiça, M. Najafzadeh, and M. Shapiro. Putting consistency back into eventual consistency. In *Proceedings of the Tenth European Conference on Computer Systems*, page 6. ACM, 2015.

[7] M. Basseville and I. V. Nikiforov. *Detection of Abrupt Changes: Theory and Application*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1993.

[8] P. A. Bernstein, V. Hadzilacos, and N. Goodman. Concurrency control and recovery in database systems. 1987.

[9] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM (JACM)*, 43(2):225–267, 1996.

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

[11] J. Du, S. Elnikety, and W. Zwaenepoel. Clock-si: Snapshot isolation for partitioned data stores using loosely synchronized clocks. In *Reliable Distributed Systems (SRDS), 2013 IEEE 32nd International Symposium on*, pages 173–184. IEEE, 2013.

[12] J. Du, D. Sciascia, S. Elnikety, W. Zwaenepoel, and F. Pedone. 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*, pages 343–354. IEEE, 2014.

[13] C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM (JACM)*, 35(2):288–323, 1988.

[14] S. Elnikety, F. Pedone, and W. Zwaenepoel. Database replication using generalized snapshot isolation. In *Reliable Distributed Systems, 2005. SRDS 2005. 24th IEEE Symposium on*, pages 73–84. IEEE, 2005.

[15] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM (JACM)*, 32(2):374–382, 1985.

[16] G. Graefe, M. Lillibridge, H. Kuno, J. Tucek, and A. Veitch. Controlled lock violation. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 85–96. ACM, 2013.

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

[18] R. Guerraoui and M. Kapalka. On the correctness of transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '08, pages 175–184, New York, NY, USA, 2008. ACM.

[19] R. Guerraoui, M. Pavlovic, and D.-A. Seredinschi. Incremental consistency guarantees for replicated objects. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, GA, 2016. USENIX Association.

[20] J. R. Haritsa, K. Ramamritham, and R. Gupta. The prompt real-time commit protocol. *IEEE Trans. Parallel Distrib. Syst.*, 11(2):160–181, Feb. 2000.

[21] P. Helland and D. Campbell. Building on quicksand. *arXiv preprint arXiv:0909.1788*, 2009.

[22] R. Jiménez-Peris, M. Patiño Martínez, B. Kemme, and G. Alonso. Improving the scalability of fault-tolerant database clusters. In *Proceedings of the 22 Nd International Conference on Distributed Computing Systems (ICDCS'02)*, ICDCS '02, pages 477–, Washington, DC, USA, 2002. IEEE Computer Society.

[23] E. P. Jones, D. J. Abadi, and S. Madden. Low overhead concurrency control for partitioned main memory databases. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 603–614. ACM, 2010.

[24] R. Kotla, M. Balakrishnan, D. Terry, and M. K. Aguilera. Transactions with consistency choices on geo-replicated cloud storage. Technical report, September 2013.

[25] T. Kraska, M. Hentschel, G. Alonso, and D. Kossmann. Consistency rationing in the cloud: Pay only when it matters. *Proceedings of the VLDB Endowment*, 2(1):253–264, 2009.

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

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

[28] C. Li, D. Porto, A. Clement, J. Gehrke, N. M. Preguiça, and R. Rodrigues. Making geo-replicated systems fast as possible, consistent when necessary. In *OSDI*, pages 265–278, 2012.

[29] Z. Li, P. Van Roy, and P. Romano. Speculative transaction processing in geo-replicated data stores. Technical Report 2, INESC-ID, Feb. 2017.

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

[31] R. Palmieri, F. Quaglia, and P. Romano. Aggro: Boosting stm replication via aggressively optimistic transaction processing. In *Network Computing and Applications (NCA), 2010 9th IEEE International Symposium on*, pages 20–27. IEEE, 2010.

[32] G. Pang, T. Kraska, M. J. Franklin, and A. Fekete. Planet: making progress with commit processing in unpredictable environments. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 3–14. ACM, 2014.

[33] A. Pavlo, E. P. Jones, and S. Zdonik. On predictive modeling for optimizing transaction execution in parallel oltp systems. *Proceedings of the VLDB Endowment*, 5(2):85–96, 2011.

[34] S. Peluso, J. Fernandes, P. Romano, F. Quaglia, and L. Rodrigues. Specula: Speculative replication of software transactional memory. In *SRDS*, pages 91–100, 2012.

[35] S. Peluso, P. Romano, and F. Quaglia. Score: A scalable one-copy serializable partial replication protocol. In *Proceedings of the 13th International Middleware Conference*, pages 456–475. Springer-Verlag New York, Inc., 2012.

[36] S. Peluso, P. Ruivo, P. Romano, F. Quaglia, and L. Rodrigues. When scalability meets consistency: Genuine multiversion update-serializable partial data replication. In *Distributed Computing Systems (ICDCS), 2012 IEEE 32nd International Conference on*, pages 455–465. IEEE, 2012.

[37] K. Ren, A. Thomson, and D. J. Abadi. An evaluation of the advantages and disadvantages of deterministic database systems. PVLDB 7(10): 821-832, 2014.

[38] J. Shute, R. Vingralek, B. Samwel, B. Handy, C. Whipkey, E. Rollins, M. Oancea, K. Littlefield, D. Menestrina, S. Ellner, et al. F1: A distributed sql database that scales. *Proceedings of the VLDB Endowment*, 6(11):1068–1079, 2013.

[39] R. S. Sutton and A. G. Barto. *Introduction to Reinforcement Learning*. MIT Press, Cambridge, MA, USA, 1st edition, 1998.

[40] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. *Managing update conflicts in Bayou, a weakly connected replicated storage system*, volume 29. ACM, 1995.

[41] A. Thomson and D. J. Abadi. The case for determinism in database systems. *Proceedings of the VLDB Endowment*, 3(1-2):70–80, 2010.

[42] A. Thomson, T. Diamond, S.-C. Weng, K. Ren, P. Shao, and D. J. Abadi. Calvin: fast distributed transactions for partitioned database systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 1–12. ACM, 2012.

[43] G. Weikum and G. Vossen. *Transactional information systems: theory, algorithms, and the practice of concurrency control and recovery*. Elsevier, 2001.

[44] P. T. Wojciechowski, T. Kobus, and M. Kokocinski. State-machine and deferred-update replication: Analysis and comparison. *IEEE Transactions on Parallel and Distributed Systems*, PP(99):1–1, 2016.

[45] C. Xie, C. Su, M. Kapritsos, Y. Wang, N. Yaghmazadeh, L. Alvisi, and P. Mahajan. Salt: Combining acid and base in a distributed database. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 495–509, 2014.