Transparent Speculation in Geo-Replicated Transactional Data Stores

Zhongmiao Li[†⋆], Peter Van Roy[†] and Paolo Romano[⋆]

[†]Université catholique de Louvain [⋆]Instituto Superior Técnico

ABSTRACT

This work presents Speculative Transaction Replication (STR), a protocol that exploits transparent speculation techniques to enhance performance of geo-distributed, partially replicated transactional data stores. In addition, we define a new consistency model, Speculative Snapshot Isolation (SPSI), that extends the semantics of Snapshot Isolation (SI) to shelter applications from the subtle anomalies that can arise from using speculative transaction processing. SPSI extends SI in an intuitive and rigorous fashion by specifying desirable atomicity and isolation guarantees that must hold when using speculative execution.

STR provides a form of speculation that is fully transparent for programmers (it does not expose the effects of misspeculations to clients). Since the speculation techniques employed by STR satisfy SPSI, they can be leveraged by application programs in a transparent way, without requiring any source-code modification to applications designed to operate using SI. STR combines two key techniques: speculative reads, which allow transactions to observe pre-committed versions, which can reduce the 'effective duration' of pre-commit locks and enhance throughput; Precise Clocks, a novel timestamping mechanism designed to enhance the chance of successful speculation.

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 STR achieves throughput gains up to 11× and latency reduction up to 10×, in workloads characterized by low inter-data center contention. Furthermore, thanks to a self-tuning mechanism that dynamically and transparently enables and disables speculation, STR offers robust performance even when faced with unfavourable workloads that suffer from high misspeculation rates.

ACM Reference Format:
Zhongmiao Li[†⋆], Peter Van Roy[†] and Paolo Romano[⋆] [†]Université catholique de Louvain [⋆]Instituto Superior Técnico . 2018. Transparent Speculation in Geo-Replicated Transactional Data Stores. In *HPDC '18: The 27th International Symposium on High-Performance Parallel and Distributed Computing, June 11–15, 2018, Tempe, AZ, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3208040.3208055

1 INTRODUCTION

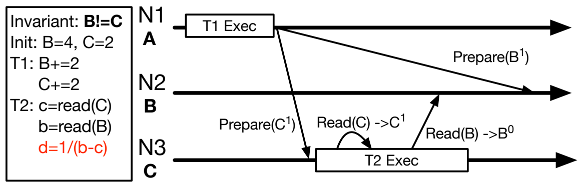
Modern large scale storage systems are increasingly deployed over geographically-scattered data centers [6, 16, 22]. Geo-replication allows storage systems to remain available even in the presence of outages affecting entire data centers and it reduces access latency by bringing data closer to clients. 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 [34], 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.

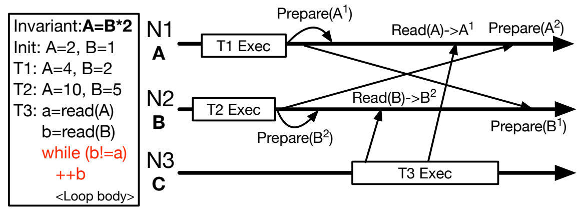
Transparent speculation. This work investigates the use of speculative processing techniques to alleviate both of the above problems. We focus on geo-distributed partially replicated transactional data stores that provide Snapshot Isolation, a widely employed consistency criterion [7, 11] (SI), and propose a novel distributed concurrency control scheme, Speculative Snapshot Isolation (SPSI), that supports a form of transparent speculative execution called *speculative reads.*

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 a *transparent speculation* technique, as misspeculations caused by it never surface to the clients and can be dealt with by simply re-executing the affected transaction.

Avoiding the pitfalls of speculation. Past work has demonstrated how the use of speculation, either transparently or requiring source-code modification [13, 15, 20, 28, 37], can



(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. T1, T2 and T3 are transactions; N1, N2 and N3 are three nodes that store data items A, B and C, respectively.

significantly enhance the performance of distributed [20, 28–30, 37] and single-site [13] transactional systems. However, these approaches suffer from several limitations:

1. Unfit for geo-distribution/partial replication. Some existing works in this area [20, 30, 37] were not designed for partially replicated geo-replicated data stores. On the contrary, they target different data models (i.e., full replication [30, 37]) or rely on techniques that impose prohibitive costs in WAN environments, such as the use of centralized sequencers to totally order transactions [20].

2. Subtle concurrency anomalies. Existing partially replicated geo-distributed transactional data stores that allow speculative reads [13, 17, 29] 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 (§5). STR shares several key design choices with state-of-the-art strongly consistent data stores [6, 7, 31], which contribute to its efficiency and scalability. These include: multi-versioning, which maximizes efficiency in read-dominated workloads [4], purely decentralized concurrency control based on loosely synchronized physical clocks [6, 7, 32], and support for partial replication [6, 21]. The key contribution of STR is its innovative, fully decentralized, concurrency control mechanism, which aims not only to ensure (SPSI-)safe speculations in a lightweight and scalable fashion, but also to enhance the chances of successful speculation via a novel transaction timestamping mechanism that we called *Precise Clocks*.
- Speculative Snapshot Isolation (SPSI), a novel consistency model that is the foundation of STR (§4). Besides guaranteeing the familiar Snapshot Isolation (SI) to *committed transactions*, SPSI provides clear and rigorous guarantees on the atomicity and isolation of the snapshots observed and produced by *executing transactions*. In a nutshell, SPSI requires an executing transaction to read data item versions committed before it started (as in SI), but it also allows to atomically observe the effects of non-conflicting transactions that originated on the same node and pre-committed before the transaction started. This shelters programmers from having to reason about complex concurrency anomalies that can otherwise arise in speculative systems.
- A lightweight yet effective self-tuning mechanism, based on a feedback control loop, that dynamically enables or disables speculation based on the workload characteristics (§5.5).
- We evaluate STR on up to nine geo-distributed Amazon EC2 data centers, using both synthetic and realistic benchmarks (TPC-C and RUBiS) (§6). Our experimental study shows that the use of transparent speculation (speculative reads) yields up to 11× throughput improvements and 10× latency reduction in a fully transparent way, i.e., requiring no compensation logic.

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. One class of geo-replicated systems [8, 39] is based on the *state-machine replication* (SMR) [23] approach, in which replicas first agree on the serialization order of transactions and then execute them without further coordination. Other recent systems [6, 7, 22] adopt the *deferred update* (DU) [19] approach, in which transactions are first locally executed and then globally certified. This approach is more scalable than SMR in update intensive workloads [19, 41] and, unlike SMR, it can seamlessly support non-deterministic transactions [33]. 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 [38]. STR builds on the DU approach and tackles its performance limitation via speculative techniques.

The property introduced in this work, SPSI, is related to PSI (Parallel Snapshot Isolation) [35], a consistency criterion that relaxes SI in order to reduce latency in geo-distributed data stores. When compared with SPSI, PSI specifies a weaker consistency criterion for final committed transactions: PSI requires that transactions read the most recent committed version of some data *only if* this is created by a transaction that originated at the same site. This allows for anomalies that are not possible in SI (called long forks [35]), and that are also excluded by SPSI, which guarantees SI-semantics for final committed transactions, i.e., they only observe the most recent committed version independently from the site in which it was originated. Further, PSI prohibits transactions from reading any version that is not final committed, which represents one of the key motivations underlying the definition of SPSI: sparing transactions from waiting for pre-commit locks to be released, while still providing rigorous consistency guarantees to shelter applications from arbitrary concurrency anomalies.

Speculation. The idea of letting transactions "optimistically" borrow, in a controlled manner, data updated by concurrent transactions has already been investigated in the past. SPECULA [30] and Aggro [26] have applied this idea to local area clusters in which data is fully replicated via total-order based coordination primitives; Jones et. al. [20] 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. However, some of these systems [26, 30] adopt a full-replication scheme, which requires all replicas to store the full dataset and apply all updates. This significantly hinders their scalability. Other systems, e.g., [20], instead, rely on the use of a global sequencer, which can become a system bottleneck and imposes unacceptably large latency in geo-distributed settings.

Other works in the distributed database literature, e.g., [13, 17, 29], 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.

Another form of speculation that strives to reduce perceived-latency by exposing preliminary results to external clients, i.e., *speculative commits*, has been explored by various works. Helland et. al. advocated the *guesses and apologies* programming paradigm [18], in which systems expose preliminary results of requests (*guesses*), but reconcile the exposed results if they are different from final results (*apologies*). A similar approach is adopted also in other recent works, like PLANET [28] and ICG [15]. Unlike STR, which is totally transparent to programmers, these approaches employ a form of external speculation, which requires source-code modification to incorporate compensation logics. Furthermore, these approaches are designed to operate on conventional storage systems, which do not support speculative reads of pre-committed data. As such, although these approaches may reduce user-perceived latency, they do not tackle the problem of reducing transaction blocking time, as STR does. We will provide experimental evidence supporting this claim in § 6.

Some of the speculative transaction processing systems mentioned above, e.g., SPECULA [30] and PLANET [28], rely on self-tuning mechanisms aimed at autonomously determining whether the use of speculation may be beneficial or not. As already mentioned, STR employs an ad-hoc self-tuning mechanism that aims at pursuing an analogous goal, i.e., dynamically enabling or disabling speculation based on the workload characteristics. More in general, there exists a large literature on self-tuning of transactional systems [9, 25, 27], which has shown the feasibility of using automatic techniques to predict and/or react timely to workload changes.

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 [2, 6]. A partition has a master replica and several slave replicas. We say that a key/partition is remote for a node, if the 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 [12]. STR integrates a classic single-master replication protocol, which assumes perfect failure detection capabilities [5]. We note, though, that it would be possible to replace the replication scheme currently employed in STR to use techniques, like Paxos [10], which require weaker synchrony assumptions.

Transaction execution model. A transaction is executed, by a process called its *coordinator*, in the node where it was originated. When it requests to commit, it undergoes a local certification phase, which checks for conflicts with concurrent transactions in the local node. If the local certification phase succeeds, we say that the transaction *local commits* and is attributed a local commit timestamp, noted LC. Next, it executes 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. Commit requests are confirmed to applications only if the transaction is final committed, which guarantees that speculative states never surface to clients. However, the versions created by a local committed transaction T can be exposed to other transactions via the *speculative read* mechanism. We say that these transactions *data depend* on T.

4 THE SPSI CONSISTENCY MODEL

We introduce Speculative Snapshot Isolation (SPSI), a consistency model that generalizes the well-known SI criterion to define a set of guarantees that shelter applications from the subtle anomalies (§Fig. 1) that may arise when using speculative techniques. Before presenting the SPSI specification, we first recall the definition of SI [40]:

- 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), and ii) local commit 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 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 property SPSI-1 is specified for *any* transaction, including the ones that eventually abort (because some other SPSI property is violated). Hence, SPSI-1 must hold throughout the execution of transactions. This 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. The same applies in case T' aborts: since SPSI-4 prohibits developing data dependencies from aborted transactions, also in this case, T must be aborted before T' is.

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.

Which anomalies does SPSI allow? SPSI provides identical guarantees to SI for final committed transactions. As for local committed and active transactions, SPSI allows for histories that would be rejected by SI, e.g., observing a version locally committed by a transaction that eventually aborts due to a conflict with some remote transaction. However, we argue that these anomalies allowed by SPSI are unharmful for applications designed to operate using SI. This is easy to show if one considers that SPSI ensures that any transaction T behaves like if it had executed under SI in a history that includes only the transactions known by the node in which T originated at the time in which T was activated. More formally, the snapshot observable by T in any SPSI-compliant history $\mathcal{H}$ is equivalent to the one that T would observe in some SI-compliant history $\mathcal{H}'$, which differs from $\mathcal{H}$ only because $\mathcal{H}'$ may omit some remote transaction concurrent with T. In other words, any snapshot observable with SPSI can be obtained via a (possible) history that would be legal using SI. Clearly, if an application works correctly with SI, i.e., it is correct with any SI-compliant history (including history $\mathcal{H}'$), the application will be also be correct when faced with history $\mathcal{H}'$ — and, thus, when executing the SPSI-compliant history $\mathcal{H}$.

Which anomalies does SPSI prevent? In Fig. 1 we have already exemplified some of the concurrency anomalies that

SPSI prevents, and which could lead applications to hang or crash. Interestingly, while analyzing the TPC-C and RUBiS benchmarks, we have identified several concurrency bugs that may arise and cause application crashes, if SPSI's guarantees are not enforced.

```
// New-Order
...
Order order;
storage->Put(order);
for (int i = 0; i < order.ol_count; i++) {
  OrderLine order_line = create_ol(order, i);
  storage->Put(order_line);
  ...
}

// Order-Status
...
Order order = storage->Read(customer.last_order);
for (int i = 0; i < order.ol_count; i++) {
  OrderLine ol = storage->Read(order.ol, i)
  // Parse throws a Null Pointer Exception if ol is null
  parse(ol);
  ...
}
```

Listing 1: Potential anomaly prevented by SPSI in TPC-C.

Listing 1 illustrates one of the anomalies we spotted in TPC-C benchmark, which involves the New Order (NO) and Order Status (OS) transactions. NO inserts a new order for a customer and then creates some number of corresponding order lines. OS fetches the identifies of the last order of a given customer, and the retrieves the corresponding order lines. In a partially-replicated setting, the order record may be stored in the node where the NO transaction was activated, but the order lines may be stored in some different node. An injudicious use of speculative reads may allow a OS transaction to read the pre-committed order record of a concurrent NO, but then allow the OS to miss the corresponding order lines (an atomicity violation that is prevented by SPSI-1). In this case, the parse method in OS would be fed with a null pointer and generate an unexpected exception, which would never occur with SI (or SPSI) and could lead to a crash of the application.

5 THE STR PROTOCOL

This section introduces the Speculative Transaction Replication (STR) protocol. For reasons of clarity, we present the design of STR incrementally. We first present a non-speculative base protocol that implements a SI-compliant transaction system. 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 place the correctness proof in our technical report [24].

5.1 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 [6, 7]. 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.

This protocol has a potential for high scalability. Unfortunately, though, in geo-distributed settings, its throughput can be severely limited by convoy effects caused by the pre-commit locks. These locks are held throughout the transactions' certification phase, which in geo-distributed data stores entail the latency of at least one inter-data center RTT —- or more if data partitions are replicated in different data-centers to allow for disaster recovery. Throughout this period, concurrent transactions attempting to read pre-committed data are conservatively blocked, which inherently limits the maximum degree of concurrency (and hence throughput) achievable by the system.

As we mentioned, the idea at the basis of STR is to tackle this problem by allowing transactions to observe pre-committed versions. Materializing this idea to build STR raised several technical challenges: guaranteeing (SPSI-)safe speculations (§ 5.2), maximizing the likelihood of successful speculation (§ 5.3) and ensuring robust performance even in adverse workload settings (§ 5.5) .

5.2 Enabling SPSI-safe speculations

Let us discuss how to extend the base protocol described above to incorporate speculative reads, while preserving SPSI semantics. 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 Fig. 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,

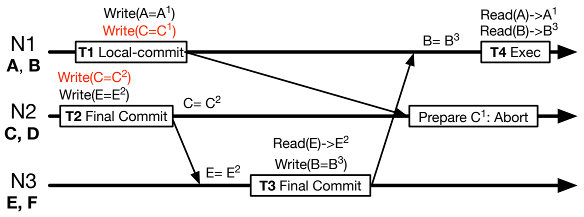


Figure 2: History exemplifying indirect conflicts between a local committed transaction, $T1$, and a final committed transaction originated at a different node, $T3$. If $T4$ included both $T1$ and $T3$ in its snapshot, it would violate SPSI property 3.

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 $T4$ when attempting to read B from $T3$, 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 remotely read 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.

5.3 Promoting successful speculation via Precise Clocks

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., *Precise Clocks*, 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 $T_1, \ldots, T_n$, 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., $T.FC\ > \max\{T_1.RS, \ldots, T_n.RS\}$. 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 property P2, STR associates to each data item an additional timestamp, called *LastReader*, which tracks the read snapshot of the most recent transaction that has 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 Clocks mechanism allows to track write-after-read dependencies among transaction at a finer granularity than the timestamping mechanism used in the base protocol — which, we recall, is also the mechanism used by non-speculative protocols like, e.g., Spanner [6] or Clock-SI [7]. Indeed, as we will show in §6, the reduction of commit timestamps achievable via Precise Clocks does not only increase the chances of successful speculation, but also reduces abort rate for non-speculative protocols.

5.4 Algorithmic definition

The pseudocode of the STR protocol is reported in Algorithms 1 and 2, which describe, respectively, the behavior of transaction coordinators and of data partitions.

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 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, 8-14). 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.

Algorithm 1: Coordinator protocol

```
1  function transaction startTx()
2      Tx.RS ← current_time()
3      Tx.Coord ← self()
4      Tx.OLCSet ← {< ⊥, ∞ >}
5      Tx.FFC ← 0
6      return Tx

7  function value read(transaction Tx, key 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 function result commitTx(transaction Tx)
       // 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     if Tx updates non-local keys
24         Tx.OLCSet.put(self(), Tx.RS)
25     send local commit requests to local replicas of updated partitions
       // Global certification
26     send prepare requests to remote master of updated partitions
27     wait receive [prepared, TS] from Tx.InvolvedReplicas
28         wait until all dependencies are resolved
29         CommitTime ← max(all received TS)
30         commit(Tx, CommitTime)
31         return committed
32     wait receive aborted
33         abort(Tx)
34         return aborted

34 function void commit(transaction Tx, timestamp CT)
35     Tx.FFC ← CT
36     Tx.OLCSet ← {< ⊥, ∞ >}
37     for Tr with data dependencies from Tx
38         if Tr.RS >= CT then
39             remove Tx from Tr's read dependency
40             Tr.OLCSet.remove(Tx)
41             Tr.FFC ← max(Tr.FFC, CT)
42         else
43             abort(Tr)
44     atomically commit Tx's local committed updates
               and remove Tx's cached updates
45     send commit requests to remote replicas of updated partitions

46 function void abort(transaction Tx)
47     abort transactions with dependencies from Tx
48     atomically remove Tx's local committed updates
49     send abort requests to remote replicas of updated partitions
```

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, 18-22). Each partition prepares the transaction if no write-write is detected, and proposes a prepare timestamp according to the Precise Clocks 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 pre-committed record to local-committed state with the local commit timestamp (Alg1, 25 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, 23-24).

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, 26). 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, 35-45). 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, 46-49).

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)

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

5  function <value, transaction> readFrom(tx Tx, key 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 function <state, timestamp> prepare(tx Tx, set 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 function void localCommit(transaction Tx, timestamp LCT,
           set 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
```

5.5 Dynamically tuning speculation

Speculative reads are based on the optimistic assumption that local-committed transactions are unlikely to experience contention with remote transactions. Although our experiments in §6 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 decides whether to enable or disable the speculative mechanisms, depending on the workload characteristics. The 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 simple feedback-driven control loop, steered by a centralized process that gathers measurements from all nodes in a periodic fashion, compares the throughput achieved with speculative reads enabled and disabled, and accordingly configures the system.

We opted for a simple and quickly converging scheme, instead of more complex approaches (e.g., based on off-line trained classifiers or more sophisticated on-line search strategies [36]), since our experimental findings confirm that, for a given workload, the decision whether or not to use speculation has a straightforward effect on throughput (no jitterlike behavior).

Our current implementation allows system administrators to initiate the self-tuning process periodically or upon request. The current self-tuning scheme could thus be naturally extended to detect statistically meaningful changes of the average input load via robust change detection algorithms, like CUSUM [3], and react to these events by re-initiating the self-tuning mechanism.

5.6 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. Like any other approach, e.g., [6, 7, 31, 32], based on 2PC, some orthogonal mechanism (typically based on replication [14]) has to be adopted to ensure the high availability of the coordinator state.

6 EVALUATION

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

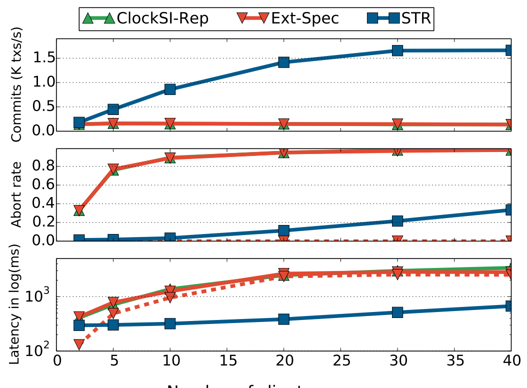
(1) What performance gains can be achieved by STR by allowing transactions to speculatively read pre-committed data?
(2) How does STR compare with systems, like PLANET [28], which employ external speculation techniques and that, unlike STR, require programmers to develop compensation logics to deal with possible misspeculations?
(3) Which workload characteristics have the strongest impact on the performance of STR?
(4) How relevant is the Precise Clocks technique, when used in conjunction with both speculative and non-speculative protocols?
(5) How effective is STR's self-tuning mechanism to ensure robust performance in presence of workloads that are not favourable to speculative techniques?

Baselines. The first baseline protocol we consider is Clock-SI [7], which we extended to support replication, as explained in §5.1. We refer to this protocol as ClockSI-Rep. ClockSI-Rep is representative of state of the art transactional protocols based on decentralized physical clocks and it provides Snapshot Isolation, namely the consistency guarantee that SPSI extends to accommodate speculation. Thus, ClockSI-Rep is an appropriate baseline to evaluate the performance gains achievable by STR thanks to the use of speculative reads and Precise Clocks.

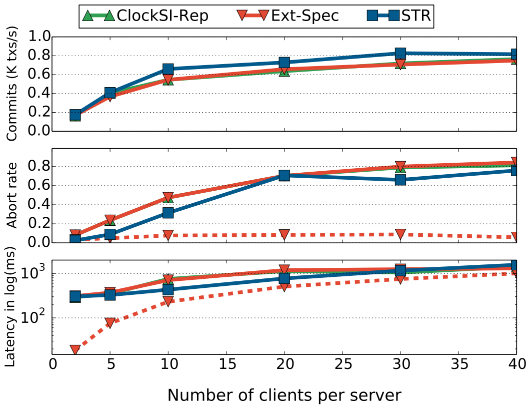
The second baseline we consider is representative of recent approaches [15, 18, 28] that propose programming models aimed to support external speculation techniques, i.e., exposing uncommitted results to clients. Supporting it comes at the cost of extra complexity for the programmers, who are forced to identify the possible concurrency anomalies that may affect their programs and develop the corresponding compensation logics (which is not needed for STR). We build this baseline, which we call Ext-Spec, by developing a variant of ClockSI-Rep that externalizes to client the results of a transaction, once it passes its local certification phase and is still undergoing its global certification phase. Note that no compensation logic is executed when using Ext-Spec: this is done for simplicity and since in the considered benchmarks, speculation can lead only to the production of incorrect replies to clients, but does not compromise the internal consistency of the server-side of the application. It should be noted that this choice actually favors Ext-Spec, as it spares this baseline from the additional overheads associated with the execution of potentially complex compensation logic.

Since Ext-Spec and ClockSI-Rep share the same (distributed) concurrency control mechanism, as we will see, they deliver very similar peak throughput, final latency and abort rate. However, Ext-Spec's use of external speculation can reduce speculative (but not final) latency, with respect to ClockSI-Rep.

Experimental setup. We implemented the baseline protocols and STR in Erlang, based on *Antidote*[1], an open-source platform for evaluating distributed consistency protocols (such as the one in [1]). More precisely, the in-memory backend of Antidote (which provides a key-value store interface) has been extended to develop fully-fledged prototypal implementations of STR and of the aforementioned baselines. The code of all protocols used in this study is publicly accessible at https://github.com/marsleezm/STR.



(a) Synth-A.



(b) Synth-B.

Figure 3: The performance of different protocols for two synthetic workloads, representative of a favourable (Synth-A) and an unfavourable (Synth-B) scenario for internal speculation. In the latency plot, we use solid lines for final latency and dashed lines for speculative latency; in the abort rate plot, we report total abort rate with solid lines and misspeculation rate with dashed lines.

Our experimental testbed is deployed across nine DCs of Amazon EC2 spanning 4 continents. 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.

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); for Ext-Spec,

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

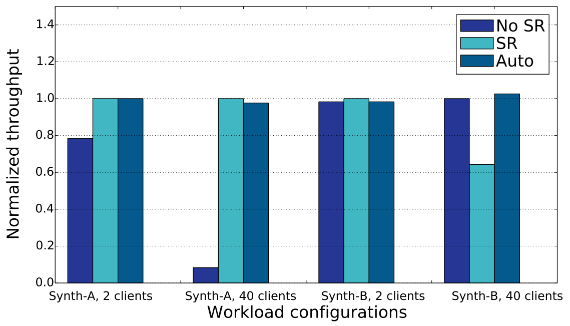


Figure 4: Normalized throughput with respect to the best performing static configuration. *No SR/SR* denote enabling/disabling statically speculative reads in STR; *Auto* denotes the use of the self-tuning technique presented in § 5.5.

we report also the *speculative latency*, which 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 Ext-Spec we also report the rate of *external misspeculation*, i.e., the percentage of transactions that were speculatively committed but finally aborted triggering the activation of some compensation logic (which we do not implement in this study, for simplicity). 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.

Unless otherwise specified, STR uses the self-tuning mechanism described in §5.5 to enable and disable the use of internal speculation. The self-tuning process gathers throughput measurements with a 10 seconds frequency. The reported results for STR refer to the final configuration identified by the self-tuning process.

6.1 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.

Transaction and data access. A transaction reads and updates 10 keys. When accessing a data partition, 10% 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 two workloads[2], which we obtain by varying the size of the hotspot sizes in the local and remote data partitions in order to synthesize two extreme scenarios that can be seen as representative of best and worst cases for internal speculation:

(1) the "best case" workload, noted Synth-A, generates very high local contention, by using a single key in the hot spots of local partitions, but very low remote contention, by using 800 keys in the hot spots of remote partitions. Due to high likelihood of local contention, transactions are very likely to speculatively read versions that were local committed by some concurrent local transaction. Since remote contention is very low, though, internal speculation is very likely to succeed.
(2) the "worst case" workload, noted Synth-B, has both very high local and remote contention, by using 10, resp. 3, keys in the hot spots of local, resp. remote, partitions. Like in workload Synth-A, transactions frequently use speculative reads, but, in this case, internal speculation is almost certainly doomed to fail due to the high remote contention.

Synth-A. Fig. 3.(a) clearly highlights the potential benefits that internal speculation can provide in favourable workload conditions. Both ClockSI-Rep and Ext-Spec fail to achieve any scalability and thrash, due to high abort rates (see middle plot), as soon as the degree of concurrency in the system grows to more than 2 clients. Conversely, STR scales almost linearly up to 20 clients and throughput saturates only at around 40 clients, achieving a 11.5× gain with respect to both baselines (which achieve very similar throughput levels). Also, the abort rate of STR is significantly lower than for the two baseline protocols. This is explicable considering that, with the baselines, any transaction T that read a key pre-committed by some concurrent transaction T' is forced to block; when T' commits, it is very likely that T' generates a commit timestamp larger the read snapshot of T, which causes T to abort. In the same scenario, though, STR would allow T to speculatively read from T'; also, the commit timestamp attributed to T' by Precise Clocks is likely to be smaller in absolute terms, and, with a higher probability than for the baselines, also smaller than the read timestamp of T. In this case, STR spares T from aborting, as well as from blocking — this allows STR not only to minimize the wasted work due to transactions' rollbacks, but also to enhance the degree of parallelism sustainable by the system.

It should be noted that since local contention dominates in this workload, most of the aborts occur during the local certification phase of transactions. Also, if transactions pass local certification, they are likely to avoid conflicts with remote transactions and, hence, commit with high probability. These considerations explain why Ext-Spec incurs an abort rate that is very similar to the one of ClockSI-Rep and to incur a very small external misspeculation rate.

As for the latency, the bottom plot shows about one order magnitude smaller final latency for STR compared to the baselines with more than 2 clients. This is due to the fact that both ClockSI-Rep and Ext-Spec are thrashing due to high contention in this load range. For analogous reasons,

[2]The evaluation results of additional workloads can be found in [24].

Techniques \ # of keys	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 Clocks/Precise Clocks; *SR* denotes that speculative reads are enabled. Throughputs reported in each column are normalized according to the throughput of 'Physical' in that column.

the speculative latency of Ext-Spec is only lower than the final latency of STR at very low load (2 clients), where the abort rate is still relatively low.

Synth-B. Fig. 3.(b) shows that, even in such an unfavourable workload for internal speculation, STR can provide robust performance that is at par with the baseline protocols. Thanks to its self-tuning capabilities, in fact, STR automatically disables the use of speculative reads for 30 or more clients, which correspond to load levels in which internal speculation has an adverse effect on performance.

This is illustrated in Fig. 4, which reports the performance achieved by STR when statically configured to enable or disable speculative reads, as well as when using the self-tuning mechanism to select between these two configurations. More in detail, the y-axis of this figure reports the throughput of each variant of STR normalized with respect to the throughput of the variant that achieves best performance for the considered workload and number of clients (on the x-axis).

By Fig. 4, we can observe that, indeed, the use of speculative reads reduces throughput by around 40% in workload Synth-B with 40 clients and that the proposed self-tuning scheme can correctly identify the optimal configuration. By this plot, we can also observe that the choice of enabling/disabling internal speculation is not only affected by the workload type — as expected, speculative reads are beneficial in Synth-A but they are not in Synth-B — but also by the level load, fixed a given workload — speculative reads do not actually penalize throughput in Synth-B with 2 clients. 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 mechanism.

Benefits and overhead of Precise Clocks. This experiment aims at quantifying the benefits stemming from the use of the Precise Clocks 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 Clocks 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 Clocks. 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 5.3, physical clock based protocols, like Clock-SI or Spanner [6, 7], tend to generate large commit timestamp, which reduces the chances that speculative reads succeed. Finally, the collective use of Precise Clocks 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 Clocks, which, we recall, requires maintaining additional metadata (a timestamp) for each accessed key. Our measurement shows that for the TPC-C and RUBiS benchmarks (§6.2), Precise Clocks requires about 9% of extra storage.

6.2 Macro benchmarks

Next, we evaluate the performance of STR by implementing two realistic benchmarks, namely TPC-C [3] and RUBiS [4]. Unlike the previous synthetic benchmarks, TPC-C and RUBiS specify several seconds of "think time" between consecutive operations issued by a client. Hence, we need to use a much larger client population to saturate the system.

TPC-C. Our TPC-C workload consists of three representative transactions: the *payment* transaction, which has very high local contention and low remote contention; *new-order* transaction, which has low local contention and high remote contention; and *order-status*, a read-only transaction. We consider three workload mixes: 5% new-order, 83% payment and 12% order-status (TPC-C A, Fig. 5.(a)); 45% new-order, 43% payment and 12% order-status (TPC-C B, Fig. 5.(b)) and 5% new-order, 43% payment and 52% order-status (TPC-C C, Fig. 5.(c)). Each server is populated with five warehouses, of which it is the master replica.

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 (ClockSI-Rep and Ext-Spec), STR achieves significant speedup especially for the 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. We see that the use of external speculation in this case barely brings any improvement on throughput over ClockSI-Rep. We also observe that the use of external speculation can significantly reduce the (speculative) latency perceived by

[3]http://www.tpc.org/tpc_documents_current_versions/pdf/tpc-c_v5.11.0.pdf
[4]http://rubis.ow2.org/

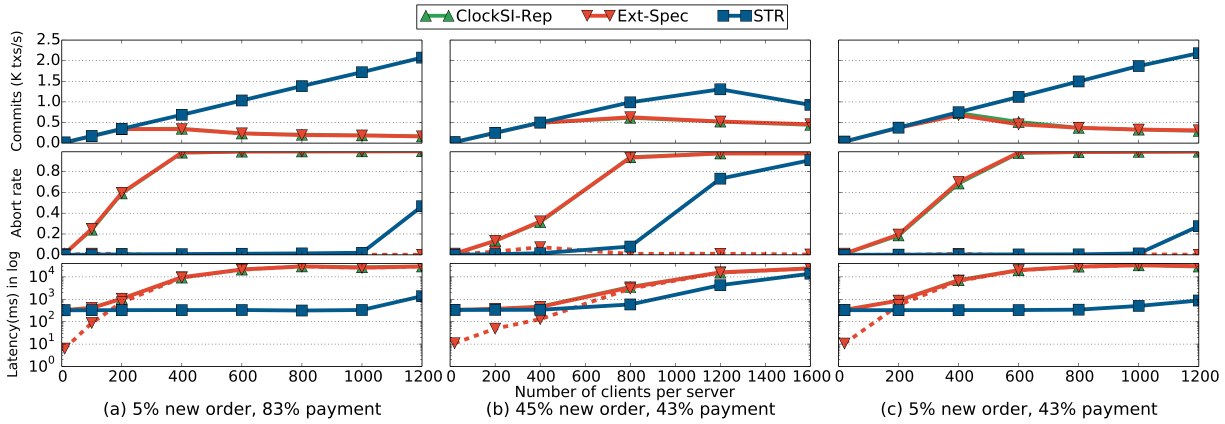


Figure 5: The performance of different protocols for three TPC-C workloads. In the latency plot, we use solid lines for final latency and dashed lines for speculative latency; in the abort rate plot, we report total abort rate with solid lines and misspeculation rate with dashed lines.

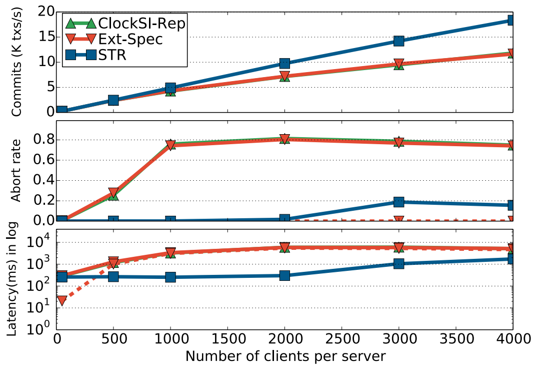


Figure 6: The performance of different protocols for RUBiS. In the latency plot, we use solid lines for final latency and dashed lines for speculative latency; in the abort rate plot, we report total abort rate with solid lines and misspeculation rate with dashed lines.

clients, but only in low load conditions. This can be explained by looking at the abort rate plots, which clearly show that, as load increases, the likelihood that external speculation is successful quickly decreases.

In fact, with larger number of clients (more than 1000 clients per server), the latency of Ext-Spec and ClockSI-Rep is on the order of 5-8 seconds, as a consequence of the high abort rate incurred by these protocols. Conversely, STR still delivers a latency of a few hundred milliseconds.

RUBiS. RUBiS 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), STR achieves about 43% higher throughput. The final latency gains of STR over the considered baselines extends up to 10× latency reduction over ClockSI-Rep and Ext-Spec. Also in this case, external speculation is effective in reducing speculative latency only at very low load levels, before loosing effectiveness and collapsing to the same performance of ClockSI-Rep.

7 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 is based on a novel consistency criterion, which we call 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. Furthermore, using STR requires no source-code modification, and for both of these reasons it is fully transparent to programmers.

STR builds on recent, highly scalable transactional protocols based on physical clocks (like Clock-SI and Google's Spanner) and extends them with a set of new speculative techniques (in particular, item-based timestamps to improve the speculation) and a self-tuning mechanism. Via an extensive experimental study, we show that STR can achieve striking gains (up to 11× throughput increase and 10× latency reduction) in workloads characterized by low inter-data center contention, while ensuring robust performance even in adverse settings.

We 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 modeling 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 for different transactions' types or at different nodes in a heterogeneous cluster.

ACKNOWLEDGEMENT

We are grateful to our shepherd Ali R. Butt and the anonymous reviewers, who provided much useful feedback that helped to improve the paper. This work is partially funded by the LightKone project in the European Union Horizon 2020 Framework Programme under grant agreement 732505, by the Erasmus Mundus Doctorate Programme under grant agreement 2012-0030 and by FCT via projects UID/CEC/50021/2013 and PTDC/EEISCR/1743/2014.

REFERENCES

[1] D. D. Akkoorath, A. Z. Tomsic, M. Bravo, Z. Li, T. Crain, A. Bieniusa, N. Preguiça, and M. Shapiro. Cure: Strong semantics meets high availability and low latency. In *ICDCS '16*, pages 405–414. IEEE, 2016.

[2] 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 '11*, volume 11, pages 223–234, 2011.

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

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

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

[6] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, et al. Spanner: Google's globally distributed database. *ACM TOCS*, 31(3):8, 2013.

[7] J. Du, S. Elnikety, and W. Zwaenepoel. Clock-SI: Snapshot isolation for partitioned data stores using loosely synchronized clocks. In *SRDS '13*, pages 173–184. IEEE, 2013.

[8] 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 *DSN '14*, pages 343–354. IEEE, 2014.

[9] S. Duan, V. Thummala, and S. Babu. Tuning database configuration parameters with ituned. *PVLDB '09*, 2(1):1246–1257, 2009.

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

[11] S. Elnikety, F. Pedone, and W. Zwaenepoel. Database replication using generalized snapshot isolation. In *SRDS '05*, pages 73–84. IEEE, 2005.

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

[13] G. Graefe, M. Lillibridge, H. Kuno, J. Tucek, and A. Veitch. Controlled lock violation. In *SIGMOD '13*, pages 85–96. ACM, 2013.

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

[15] R. Guerraoui, M. Pavlovic, and D.-A. Seredinschi. Incremental consistency guarantees for replicated objects. In *OSDI '16*, GA, 2016. USENIX Association.

[16] A. Gupta, F. Yang, J. Govig, A. Kirsch, K. Chan, K. Lai, S. Wu, S. G. Dhoot, A. R. Kumar, A. Agiwal, et al. Mesa: Geo-replicated, near real-time, scalable data warehousing. *PVLDB '14*, 7(12):1259–1270, 2014.

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

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

[19] R. Jiménez-Peris, M. Patiño Martínez, B. Kemme, and G. Alonso. Improving the scalability of fault-tolerant database clusters. In *ICDCS '02*, pages 477–484. IEEE, 2002.

[20] E. P. Jones, D. J. Abadi, and S. Madden. Low overhead concurrency control for partitioned main memory databases. In *SIGMOD '10*, pages 603–614. ACM, 2010.

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

[22] T. Kraska, G. Pang, M. J. Franklin, S. Madden, and A. Fekete. Mdcc: Multi-data center consistency. In *Eurosys '13*, pages 113–126. ACM, 2013.

[23] L. Lamport. The part-time parliament. *ACM TOCS*, 16(2):133–169, May 1998.

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

[25] J. Paiva, P. Ruivo, P. Romano, and L. Rodrigues. Autoplacer: Scalable self-tuning data placement in distributed key-value stores. *ACM TAAS*, 9(4):19, 2015.

[26] R. Palmieri, F. Quaglia, and P. Romano. Aggro: Boosting stm replication via aggressively optimistic transaction processing. In *NCA '10*, pages 20–27. IEEE, 2010.

[27] R. Palmieri, F. Quaglia, P. Romano, and N. Carvalho. Evaluating database-oriented replication schemes in software transactional memory systems. In *IPDPSW '10*, pages 1–8. IEEE, 2010.

[28] G. Pang, T. Kraska, M. J. Franklin, and A. Fekete. Planet: making progress with commit processing in unpredictable environments. In *SIGMOD '14*, pages 3–14. ACM, 2014.

[29] A. Pavlo, E. P. Jones, and S. Zdonik. On predictive modeling for optimizing transaction execution in parallel oltp systems. *PVLDB '11*, 5(2):85–96, 2011.

[30] S. Peluso, J. Fernandes, P. Romano, F. Quaglia, and L. Rodrigues. Specula: Speculative replication of software transactional memory. In *SRDS '12*, pages 91–100, 2012.

[31] S. Peluso, P. Romano, and F. Quaglia. Score: A scalable one-copy serializable partial replication protocol. In *Middleware '12*, pages 456–475. Springer-Verlag New York, Inc., 2012.

[32] S. Peluso, P. Ruivo, P. Romano, F. Quaglia, and L. Rodrigues. When scalability meets consistency: Genuine multiversion update-serializable partial data replication. In *ICDCS '12*, pages 455–465. IEEE, 2012.

[33] K. Ren, A. Thomson, and D. J. Abadi. An evaluation of the advantages and disadvantages of deterministic database systems. PVLDB '14, 2014.

[34] 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. *PVLDB '13*, 6(11):1068–1079, 2013.

[35] Y. Sovran, R. Power, M. K. Aguilera, and J. Li. Transactional storage for geo-replicated systems. In *SOSP '11*, pages 385–400. ACM, 2011.

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

[37] 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. SOSP '95, pages 172–182, New York, NY, USA, 1995. ACM.

[38] A. Thomson and D. J. Abadi. The case for determinism in database systems. *PVLDB '10*, 3(1-2):70–80, 2010.

[39] A. Thomson, T. Diamond, S.-C. Weng, K. Ren, P. Shao, and D. J. Abadi. Calvin: fast distributed transactions for partitioned database systems. In *SIGMOD '12*, pages 1–12. ACM, 2012.

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

[41] P. T. Wojciechowski, T. Kobus, and M. Kokocinski. State-machine and deferred-update replication: Analysis and comparison. *IEEE TPDS*, PP(99):1–1, 2016.