Serializable Snapshot Isolation for Replicated Databases in High-Update Scenarios

Hyungsoo Jung[†]
University of Sydney

Hyuck Han[*]
Seoul National University

Alan Fekete[†]
University of Sydney

Uwe Röhm[†]
University of Sydney

[†]{firstname.lastname}@sydney.edu.au [*]hhyuck@dcslab.snu.ac.kr

ABSTRACT

Many proposals for managing replicated data use sites running the Snapshot Isolation (SI) concurrency control mechanism, and provide 1-copy SI or something similar, as the global isolation level. This allows good scalability, since only *ww*-conflicts need to be managed globally. However, 1-copy SI can lead to data corruption and violation of integrity constraints [5]. 1-copy serializability is the global correctness condition that prevents data corruption. We propose a new algorithm *Replicated Serializable Snapshot Isolation* (RSSI) that uses SI at each site, and combines this with a certification algorithm to guarantee 1-copy serializable global execution. Management of *ww*-conflicts is similar to what is done in 1-copy SI. But unlike previous designs for 1-copy serializable systems, we do not need to prevent all *rw*-conflicts among concurrent transactions. We formalize this in a theorem that shows that many *rw*-conflicts are indeed false positives that do not risk non-serializable behavior. Our proposed RSSI algorithm will only abort a transaction when it detects a well-defined pattern of two consecutive *rw*-edges in the serialization graph. We have built a prototype that integrates our RSSI with the existing open-source Postgres-R(SI) system. Our performance evaluation shows that there is a worst-case overhead of about 15% for getting full 1-copy serializability as compared to 1-copy SI in a cluster of 8 nodes, with our proposed RSSI clearly outperforming the previous work [6] for update-intensive workloads.

1. INTRODUCTION

In 1996, a seminal paper by Gray et al. [15] showed that there were performance bottlenecks that limited scalability of all the then-known approaches to managing replicated data. These approaches involved a transaction performing reads at one site, performing writes at all sites, and obtaining locks at all sites. This led to a busy research area, proposing novel system designs to get better scalability. In 2010, in a reflection [19] on winning a Ten-Year-Best-Paper award, Kemme and Alonso identified some important aspects of this work. Among these is the value of Snapshot Isolation (SI) rather than serializability. SI is widely available in DBMS engines like Oracle DB, PostgreSQL, and Microsoft SQL Server. By using SI in each replica, and delivering 1-copy SI as the global behavior, most recent proposals have obtained improved scalability, since local SI can be combined into (some variant of) global 1-copy SI by handling *ww*-conflicts, but ignoring *rw*-conflicts. This has been the dominant replication approach, in the literature [10, 11, 12, 13, 23, 27].

While there are good reasons for replication to make use of SI as a local isolation level (in particular, it is the best currently available in widely deployed systems including Oracle and PostgreSQL), the decision to provide global SI is less convincing. Global SI (like SI in a single site) does not prevent data corruption or interleaving anomalies. If data integrity is important to users, they should expect global (1-copy) serializable execution, which will maintain the truth of any integrity constraint that is preserved by each transaction considered alone.

Thus in this paper, we propose a new replication algorithm that uses sites running SI, and provides users with a global serializable execution.

Recently, Bornea et al. [6] proposed a replica management system that uses sites running SI, and provides users with a global serializable execution. They use an essentially optimistic approach (following the basic principles from [17]) in which each transaction is performed locally at one site, and then (for update transactions) the writeset and readset are transmitted (by a totally ordered broadcast) to remote sites where a certification step at the end of the transaction can check for conflicts between the completing transaction and all concurrent committed transactions; if a conflict is found, the later transaction is aborted. All the designs that offer SI as global isolation level, must do essentially the same checks to prevent *ww*-conflicts; the extra work for 1-copy serializability lies in preventing *rw*-conflicts. Bornea et al. follow this approach in a middleware-based design, with readsets calculated by the middleware from analysis of SQL text, in order to reduce transmission cost. We refer to this conflict management approach, of aborting an update transaction when it has a *rw*- or *ww*-conflict with a committed concurrent transaction, as the "conflict-prevention with read-only optimisation" approach (abbreviated as CP-ROO). Our paper offers a more subtle approach, that looks at pairs of consecutive conflict edges in order to reduce the number of unnecessary aborts; as a trade-off however, we must also certify read-only transactions, and so our technique is suited to

Proceedings of the VLDB Endowment, Vol. 4, No. 11
Copyright 2011 VLDB Endowment 2150-8097/11/08... $ 10.00.

cases of high update rate (as we demonstrate in the evaluation). These cases are not well treated by existing replication strategies which focus instead on read-mostly environments, and they scale badly when updates are common.

A similar issue was previously studied in single-site DBMS concurrency control. Theory from Adya [1] as extended by Fekete et al. [14] showed that checking each *rw*-conflict in an SI-based system gives many unnecessary aborts ("false positives") in situations where non-serializable execution will not arise even in presence of a *rw*-conflict; instead, non-serializable execution only arises from a pair of consecutive *rw*-conflicts within a cycle in the serialization graph. Cahill et al. [7] used this to propose Serializable Snapshot Isolation (SSI), a concurrency control mechanism that is a modification of an SI-based (single site) engine. SSI guarantees serializable execution, with performance close to that of SI itself (note that a pair of consecutive edges is much less frequent than a single edge, so SSI has far fewer aborts than a standard optimistic approach). In this paper we do a similar thing for a replicated system, with SI as the mechanism in each site, and 1-copy serializability as the global property of executions. We prevent *ww*-conflicts in the same way as 1-copy SI designs, and we prevent certain pairs of consecutive *rw*-conflicts.

Contributions of This Work. We prove a **new theorem** that gives a sufficient condition for 1-copy serializable execution, based on absence of *ww*-conflicts between concurrent transactions, and absence of certain patterns of consecutive *rw*-conflicts. We propose a **concurrency control algorithm** that applies the theorem, to abort transactions during a certification step when these situations are found. We design and implement **a prototype system** called Postgres-RSSI, using this algorithm, as a modification of the existing Postgres-R(SI) open source replication system. Our prototype uses a combination of existing ideas, to be quite efficient. In particular, we use a carefully-arranged mechanism for total-order broadcast of the transaction completion messages, including the writesets, and dependency information about *rw*-conflicts. That is, we avoid sending potentially large readsets in messages. We **measure the performance** of our prototype and show that there is not too much degradation compared to (non-serializable) designs like Postgres-R(SI), and that (when updates transactions are frequent) we do significantly better than a design that prevents all *rw*-conflicts between update transactions.

Paper Road Map. In Section 2 we summarize related research. Section 3 has the theorem that is a sufficient condition for global serializability, and we give the corresponding concurrency control algorithm; in Section 4 we describe the details of the efficient certification, without transmitting readsets. Section 5 presents the implementation details of our prototype, and in Section 6 we evaluate its performance. Section 7 concludes.

Acknowledgements. This work was supported by research grant DP0987900 from the Australian Research Council.

2. BACKGROUND

The field of transaction management, and its subfield of replica management, is extensive; authoritative coverage can be found in the texts by Weikum and Vossen [26] and by Kemme et al. [20], respectively. This section points only to some especially close work on which we build.

Snapshot Isolation. Snapshot Isolation (SI) is a method of multiversion concurrency control in a single DBMS engine that ensures non-blocking reads and avoids many anomalies. SI was described in Berenson et al. [5] and it has been implemented in several production systems including Oracle DB, PostgreSQL and Microsoft SQL Server. When SI is the concurrency control mechanism, and each transaction T_i begins its execution, it gets a begin timestamp b_i. Whenever T_i reads a data record x, it does not necessarily see the latest value written to x; instead T_i sees the version of x that was produced by the last to commit among the transactions that committed before T_i started and also modified x (there is one exception to this: if T_i has itself modified x, it sees its own version). Owing to this rule, T_i appears to run on a snapshot of the database, storing the last committed version of each record at b_i. SI also enforces another rule on writes, called the First-Committer-Wins (FCW) rule: two concurrent transactions that both modify the same data record cannot both commit. This prevents lost updates in a single site DBMS.

The theoretical properties of SI have been studied extensively. Berenson et al. [5] showed that SI allows some non-serializable executions. Adya [1] showed that in any non-serializable execution of SI, the serialization graph has a cycle with two consecutive *rw*-edges. This was extended by Fekete et al. [14], who found that any non-serializable execution had consecutive *rw*-edges where the transactions joined by the edge were concurrent with one another. Our new theorem is similar in format to those in [1, 14], but with different side-conditions that are suited to a replicated system.

This theory has been used to enforce serializability based on SI mechanisms. In Fekete et al. [14] and in Alomari et al. [2], compile-time analysis of conflicts was used to modify the application code so that all executions are serializable. Cahill et al. [7] instead proposed a modified mechanism, called Serializable Snapshot Isolation (SSI), that can be coded in the DBMS engine; this checks at run-time for the consecutive *rw*-edges, and aborts a transaction as needed to prevent non-serializable executions. Being in a single DBMS, [7] can detect conflict edges directly in lock tables, while (being in a system with replicas) we need to transmit information about writesets from node to node.

Replication. The traditional approach to replica management involves read one copy, write all copies (ROWA). Gray et al. [15] identified a series of performance hazards that locking-based database replication could face. This inspired much work on system designs to reduce the performance bottlenecks. These are classified by three design criteria; whether an approach is based on changes to the database kernel or by middleware that uses (nearly) unmodified single node DBMS engines for the replicas; whether updates are allowed at any site, or only at a "master" primary site, and whether transaction propagation to other replicas is done eagerly (as part of commit processing) or lazily after the commit. Kemme et al. [18] proposed a kernel-based, multi-master and eager replication protocol, which makes use of an available group communication system to guarantee a total order delivery which allows sites to remain consistent (so, 1-copy serializability can be ensured). Improved performance came from doing the database processing at one site only, and applying efficient primary-keyed write operations at other sites [17], with a certification to prevent conflicts, that was done consistently at all sites. Using sites that provide SI as concurrency control, and giving one-copy SI rather than serializability, allows a great improvement in performance, since only *ww*-conflicts need be considered in certification. This was done by Wu et al. [27], and the implementation of this, called Postgres-R(SI) has continued as an open-source project. It serves as the baseline of our prototype system.

Most recent work has focused on middleware designs, as explained in [19]. This began with MIDDLE-R [24] which supports 1-copy serializability for sites using locking, depending on a priori knowledge on which data partitions a particular application program would access. Lin et al. [23] provide 1-copy SI as the global isolation level, for sites using SI locally; this approach allows the middleware to ignore the readsets. Various session properties are given by different designs, for example Elnikety et al. [13] provide GSI, in which the snapshot of transaction T may be earlier than that containing all transactions that commit before T starts. Improved performance came with moving durability to the middleware [11], and adding load balancing and filtering [12]. Daudjee et al. [10] proposed a lazy-propagation design that ensures a strong session property; they evaluated its correctness using simulation and compared the costs to designs giving other variants of SI. A simpler design is possible with a master-slave design as in Ganymed [25] where all updates happen at one primary site; this has other benefits in allowing heterogeneity in the hardware and software of the slaves.

A new work by Bornea et al. [6] achieves 1-copy serializability from sites with SI locally. This is a middleware design, where the middleware uses program analysis to determine the predicates involved in SELECT statements, as a proxy for the readsets. The certification step, like that in [24], uses an approach where any detected *ww*-edge or *rw*-edge among update transactions leads to the abort of one of the transactions involved. The main differences between our new work and [6] are the overall architecture (kernel modifications with a broadcast that is integrated with the dependency checks, rather than a middleware layer that adds a separate certifier site and a proxy to act as a veneer over each unmodified replica DBMS), the way readsets are obtained (modification of the kernel to remember which records are read, rather than predicate analysis of SQL code), and the cases where a transaction must abort to avoid non-serializable execution (two consecutive *rw*-conflict edges among arbitrary transactions, rather than a single conflict edge among update transactions).

3. ENSURING 1-COPY SERIALIZABILITY

In the following, we describe a concurrency control algorithm that provides 1-copy serializable executions for a replicated system where each node runs at SI. In particular, we developed a new theorem that gives a sufficient condition for 1-copy serializable execution. We also give the high-level account of our algorithm, that uses the theorem as its foundation.

3.1 Notation

For simplicity, we assume that all events in the distributed system occur in a total order; effectively, we assume that some logical clock is operating. Thus the history of the system can be given as a sequence of events. Since the individual nodes are using SI, they keep versions of each item. We use the notation that X_i represents the version of item X produced by user transaction T_i (during the user-level event $W_i(X_i)$). When T_i reads item X, it will return the value from some version, produced by some transaction. If the read by T_i returns the version produced by T_j, we write this event as $R_i(X_j)$. We also include in the history the beginning and the completion (commit or abort) event for each transaction; we write b_i for the event when T_i starts executing, c_i for the commit point of T_i, and a_i for its abort point. The version order for versions of item X will be the order of the commit events of the transactions that produce the versions.

We associate three important numbers, which we call timestamps, with each transaction. The timestamps in our system are counters, which “tick” once at each transaction that attempts to commit. We use the symbol $\prec$ for “less than” among timestamps (or $\preceq$ when equality is allowed). Because our architecture sends transaction completion through a total-order broadcast mechanism, the commit-timestamp of any transaction is just its position in the broadcast total order. We write $commit(T_i)$ for the commit-timestamp of T_i. The begin-timestamp $begin(T_i)$ captures the most recent commit that has occurred at the site where T_i runs, when T_i starts to execute. Because each site uses SI, this means that each read in T_i sees the version that includes all writes by transactions whose commit-timestamp is up to, and including, the begin-timestamp of T_i. Clearly $begin(T_i) \prec commit(T_i)$.

We also define a latest-snapshot-version timestamp $lsv(T_i)$, which gives the highest commit-timestamps among the transactions whose versions of items are accessed by T_i. If we simplify by ignoring blind writes, so each transaction reads an item before writing it, this is just the maximum commit-timestamp among the versions that are read by T_i. Notice that $lsv(T_i) \preceq begin(T_i)$, but the equality may or may not occur, depending on whether there is any item in the intersection of $readset(T_i)$ and $writeset(T_j)$, where T_j is the transaction such that $commit(T_j) = begin(T_i)$, that is, T_j is the most recently committed transaction at the node when T_i starts running.

As usual in multiversion concurrency theory, we consider a serialization graph $SG(h)$ with three types of dependency edges. We have a *ww*-dependency edge $T_i \xrightarrow{ww} T_j$ when there is a data item X, such that T_i writes a version X_i, and T_j writes a version X_j that is later in the version order than X_i. We have a *wr*-dependency edge $T_i \xrightarrow{wr} T_j$ when there is a data item X such that T_j contains the event $R_j(X_i)$, so that T_j reads a version produced by T_i. Finally, we have an anti-dependency (also called *rw*-dependency) $T_i \xrightarrow{rw} T_j$, when there is an item X such that T_i reads a version X_k which is earlier in the version order than X_j written by T_j. That is, any serialization must place T_i ahead of T_j, since T_i did not see the effects of T_j on item X.

Within the category of anti-dependency edges, we give special attention to those which form what we call a descending structure. This is formed in certain cases when there are three transactions connected by successive anti-dependency edges.

Definition 3.1 (Descending structure). *Consider three transactions T_p, T_f, and T_t. We say that these form a descending structure, with T_p as peak-transaction, T_f as* *follow-transaction, and T_t as trailing transaction, when there are the following relationships:*

- *there are anti-dependency edges* $T_p \xrightarrow{rw} T_f$*, and* $T_f \xrightarrow{rw} T_t$*,*
- $lsv(T_f) \preceq lsv(T_p)$*, and*
- $lsv(T_t) \preceq lsv(T_p)$

3.2 Theory

Our concurrency control algorithm is based on enforcing a sufficient condition for 1-copy serializability, expressed in the following theorem.

THEOREM 3.1. *Let h be a history over a set of transactions, such that one can assign to each transaction T_i an event $snapshot_i$ that precedes b_i, obeying the following three conditions.*

- **Read-from-snapshot** *If h contains $R_i(X_j)$ then h contains c_j preceding $snapshot_i$, and whenever h contains $W_k(X_k)$ for some committed transaction $T_k \neq T_j$, then either c_k precedes c_j, or else c_k follows snapshot (T_i).*
- **First-committer-wins** *If h contains both $W_i(X_i)$ and $W_j(X_j)$, then it is not possible that c_i occurs between $snapshot(T_j)$ and c_j.*
- **No-descending-structure** *The serialization graph $SG(h)$ does not contain a descending structure among transactions that all commit.*

Then h is 1-copy serializable.

For a proof of Theorem 3.1, readers are referred to Appendix A, but we remark on how Theorem 3.1 relates to other results.

The read-from-snapshot condition deals with *wr*-edges. it holds automatically in any system that runs SI at each replica, sends each transaction to be performed at one replica, and then propagates the updates to other replicas in commit-order (whether this order comes from a total-order broadcast, or by having all updates at a single primary site). This is because we can choose the event $snapshot(T_i)$ to immediately follow the commit of the last transaction whose commit was received at the appropriate node before T_i begins its execution. That is, the timestamp $begin(T_i)$ is the highest commit-timestamp of any transaction that commits before the event $snapshot_i$.

The first-committer-wins condition deals with *ww*-edges. The isolation level GSI (generalized snapshot isolation, [13]) is defined exactly as the combination of condition read-from-snapshot and first-committer-wins; thus systems like Postgres-R(SI) or Tashkent, which offer GSI, do so by enforcing the first-committer-wins check in a certification step, based on knowledge of each transaction's writeset.

To ensure serializability, one must also worry about readsets and how they lead to *rw*-edges. The recent work [6] uses a condition called "no read impact" that constrains each *rw*-edge. It states that if h contains $R_j(X_i)$, then it is not possible that c_i occurs between $snapshot(T_j)$ and c_j. The power of our no-descending-structure condition is that it looks for pairs of consecutive *rw*-edges, which should be much less frequent than single *rw*-edges. In this regard, note that Adya [1] already showed that the absence of consecutive *rw*-edges is sufficient for serializability in systems based on SI (and the result holds immediately for GSI as well); the novelty of condition no-descending-structure lies in the extra aspects concerning the *lsv* timestamps.

3.3 Concurrency Control Algorithm

One can use Theorem 3.1 for a family of concurrency control algorithms that each achieves 1-copy serializability from replicas that provide SI. We follow the standard model for ROWA-replication, with each transaction submitted at one node, and performed locally there; then the modifications made are sent to all replicas, in a consistent order. At each replica, a certification check occurs, and if the transaction can commit, its modifications are installed through a "remote transaction". Certification is done using Theorem 3.1: a transaction will be allowed to commit provided that first-committer-wins and no-descending-structure are both valid.[1] To implement this idea, we need to make sure that at certification, each node has the information it needs. For checking first-committer-wins, the node needs information on the writeset of the current transaction, and the writesets of the previously committed ones. This is the same check as done for replication that gives GSI as the isolation level (as in [27, 13]), and the information is naturally available since the writesets need to be propagated anyway for installation at remote nodes.

The core challenge, in finding an efficient implementation of an algorithm in the family based on Theorem 3.1, is the management of readset information and *lsv*-timestamps, so that certification can check for a descending structure. We have modified the kernel to track the set of ids of items that are read by each transaction.[2] A naive approach would capture the readset and propagate it in the broadcast completion message (just like the writeset) but this is unrealistic in practice, since readsets may be extremely large, and the cost of sending them around would be excessive. For example, consider a transaction which calculates an aggregate over a large table; the readset is every row in the table. Instead, the algorithm that we present in this paper keeps readset information only at the local node where a transaction runs.

As each node has readsets for the transactions that are local at a node, it can determine *rw*-edges involving any other transaction when learning of the completion of that other transaction. It is this set of edges that we will share with each node for use in certification. This means that we do not simply use a black-box total-order broadcast to propagate a transaction completion message; instead, we integrate sharing the *rw*-edges with the ordering and broadcast, so that as part of the communication, the *rw*-edges can be collected from all nodes. Fortunately, the LCR broadcast protocol [16] visits each node around a ring structure, when information can be collected, before visiting each node again to deliver the message in order. Thus we have designed our system with a component GDCP, described in detail be-

[1] We remark that when a descending structure is found, the transaction that aborts will always be whichever completes last among the three; this could be the peak, the follow or the tail transaction. This contrasts with the centralized algorithm of Cahill et al. [7] which tries to abort the middle "pivot" of the three.

[2] In our implementation, we only track records that are actually read; to prevent phantoms, one will also need to include next-keys when scanning.

low, that follows LCR and collects *rw*-edge information (as well as *lsv* information). When the certification check occurs each node has the information needed, and so each node can test for a descending structure, and each will reach the same decision to commit (or to abort) a transaction.

4. GLOBAL DEPENDENCY CHECKING

One of challenges in implementing RSSI lies in the consistent retrieval of global dependency information among concurrent transactions. We approach this problem from a different perspective and propose a *global dependency checking protocol* (GDCP). A standalone process that carries out RSSI-related work is deployed, called a replication manager. GDCP has two notable features; it does *not* need to propagate an entire readset and it also reduces the amount of comparisons done at each node in dependency checking.

The basic principle is, for a local transaction T_i, a replication manager sends a writeset of T_i, along with a list of transaction IDs that have an incoming anti-dependency from T_i, instead of sending the entire readset. A replication manager checks only the presence of an outgoing anti-dependency from its local readsets to writesets of other transactions, and it piggybacks that information on writesets and forwards writesets to other replicas. This rather simple technique not only makes all replicas retrieve complete dependency information, but also is pivotal in reducing redundant dependency checking.

To ensure data consistency, replicated databases deploy uniform total order broadcasting. Our choice is to implement the LCR protocol [16] inside a replication manager. LCR is especially targeted for replicated databases and can be better for this than generic group communication systems, Spread [3] and JGroup [4].

4.1 GDCP

GDCP implements LCR straightforwardly and builds the dependency checking protocol on top of LCR. Algorithm 1 describes the pseudo-code of GDCP, and implicit in this pseudo-code is the manipulation of vector clocks for the uniform total order delivery (refer to [16] for details). The important invariant that GDCP preserves, is to acquire all anti-dependency information created from transactions with an earlier total order than a transaction checked. As a step toward such an invariant, GDCP requires four types of transactional logs, two of which are subsets of the other sets. As shown in Algorithm 1, $\mathcal{T}_c$ is a transactional log for all committed transactions including both local and remote transactions, while $\mathcal{T}_p$ is a log for all pending transactions. $\mathcal{T}_{l,c}$ and $\mathcal{T}_{l,p}$ are subsets of $\mathcal{T}_c$ and $\mathcal{T}_p$, respectively. Both point to local committed/pending transactions. For discussion on dealing with garbage logs, readers are referred to Appendix C.2.3.

The first stage of GDCP starts when a replication manager receives a readset (R_i) and a writeset (W_i) from a local database. Using these sets, a replication manager creates a protocol message that should convey the following contents: a writeset, outgoing anti-dependency information from R_i to the writesets of committed/pending transactions (i.e., $\mathcal{D}_{f,c}$ and $\mathcal{D}_{f,p}$ from $\mathcal{T}_c$ and $\mathcal{T}_p$) and incoming anti-dependency information from the readsets of local committed/pending transactions to W_i (i.e., $D_{t,c}$ and $\mathcal{D}_{t,p}$ from $\mathcal{T}_{l,c}$ and $\mathcal{T}_{l,p}$). Circulating $\mathcal{D}_{f,c}$ and $\mathcal{D}_{f,p}$ in a ring topology informs other replicas of T_i's outgoing anti-dependency information, while we collect incoming anti-dependency information from other replicas by distributing $D_{t,c}$ and $\mathcal{D}_{t,p}$.

In a protocol message, carriers for incoming dependency information have different data types (i.e., a pair $D_{t,c}$ and a set $\mathcal{D}_{t,p}$). In principle, the latest *lsv* of a transaction having a *rw*-edge to T_i is vital to our SSI theory, so a pair type of $D_{t,c}$ satisfies the principle by carrying one pair from all committed transactions. By contrast, pending transactions are not confirmed yet to be delivered (some are aborted by our SSI concurrency control), and this requires us to keep all candidate pairs in a set $\mathcal{D}_{t,p}$ until their delivery decision is made. Any transaction belonging to above sets (or the pair) must have an earlier commit timestamp (or vector clock) than the current transaction (line 1-5 of **gdcpBroadcast**). Once the protocol message is created, we check presence of a descending structure. If the current transaction is involved in a non-serializable case with committed transactions, then it should be aborted immediately. Otherwise, a replication manager starts forwarding the protocol message to its successor (line 6-10 of **gdcpBroadcast**).

A replication manager, upon receiving a protocol message for a transaction T_i from its predecessor, executes different routines based on the contents of the received protocol message. For convenience, we put a pair of special characters $(\perp, \perp)$ in $D_{t,c}$ whenever we detect a cycle.

- **Remote writeset without a cycle**. If a protocol message has a remote writeset without a cycle mark (i.e., $D_{t,c} \neq (\perp,\perp)$), a replication manager first checks whether W_i has either an anti-dependency from the readsets of local committed transactions, or any cycle with them. If a replication manager finds an anti-dependency from local committed transactions, then it updates that information in $D_{t,c}$ only if the *lsv* of its local committed transaction is later than the existing one, because the latest *lsv* of T_{in} is the most important criterion in detecting presence of a descending structure. Besides checking dependency, if a replication manager detects a cycle, then it marks that information on $D_{t,c}$ with $(\perp,\perp)$ and stops checking (line 3-8 of the first **Rreceive**). When checking an anti-dependency from local pending transactions, we only consider transactions that have an earlier commit order (vector clock) than that of T_i as by the invariant enacted. Whenever a replication manager detects an anti-dependency, it appends that information to $\mathcal{D}_{t,p}$ (line 9 of the first **Rreceive**).

  After completing dependency checking, a replication manager reflects all incoming and outgoing dependency information of T_i ($\mathcal{D}_{f,c}$, $\mathcal{D}_{f,p}$, $D_{t,c}$ and $\mathcal{D}_{t,p}$) to its local pending log $\mathcal{T}_p$, updating its own global dependency information. Then, it forwards the protocol message to its successor. Consequently, the protocol message conveys two types of anti-dependency information, one for the readset and the other for the writeset (line 10-11 of the first **Rreceive**).

- **Remote writeset with a cycle**. If a protocol message has a remote writeset with a cycle mark, then a replication manager forwards the protocol message to its successor immediately (line 12-13 of the first **Rreceive**).

- **Local writeset**. When a replication manager receives its own protocol message, it starts the second circulation of the protocol message with "Decide" to let other replicas make a decision for T_i based on the obtained depen-

Algorithm 1 Pseudo-code of GDCP

```
$W_i$ and $R_i$: a writeset and a readset of transaction $T_i$.
$\mathcal{T}_c$: a log for committed transactions.
$\mathcal{T}_p$: a log for pending transactions.
$\mathcal{T}_{l,c}$: a log for local committed transactions ($\mathcal{T}_{l,c} \subset \mathcal{T}_c$).
$\mathcal{T}_{l,p}$: a log for local pending transactions ($\mathcal{T}_{l,p} \subset \mathcal{T}_p$).
$\mathcal{D}_{f,c}$: a set of $(TID, lsv)$ pairs in $\mathcal{T}_c$, having $\overset{rw}{\dashrightarrow}$ from $T_i$.
$\mathcal{D}_{f,p}$: a set of $(TID, lsv)$ pairs in $\mathcal{T}_p$, having $\overset{rw}{\dashrightarrow}$ from $T_i$.
$D_{t,c}$: a pair of $(TID, lsv)$ in $\mathcal{T}_{l,c}$, having $\overset{rw}{\dashrightarrow}$ to $T_i$.
$\mathcal{D}_{t,p}$: a set of $(TID, lsv)$ pairs in $\mathcal{T}_{l,p}$, having $\overset{rw}{\dashrightarrow}$ to $T_i$.
$\mathcal{SSI}_{non}$: non-serializable cases, i.e., {descending structure, $\overset{ww}{\dashrightarrow}$}.

procedure gdcpBroadcast($W_i$, $R_i$)
1:  $\mathcal{D}_{f,c} \leftarrow \{(j, lsv_j) | (T_i \overset{rw}{\dashrightarrow} T_j) \bigwedge T_j \in \mathcal{T}_c\}$;
2:  $\mathcal{D}_{f,p} \leftarrow \{(j, lsv_j) | (T_i \overset{rw}{\dashrightarrow} T_j) \bigwedge T_j \in \mathcal{T}_p\}$;
3:  $lsv_m \leftarrow max\{lsv_j | (T_j \overset{rw}{\dashrightarrow} T_i) \bigwedge T_j \in \mathcal{T}_{l,c}\}$;
4:  $D_{t,c} \leftarrow (m,\ lsv_m)$;
5:  $\mathcal{D}_{t,p} \leftarrow \{(j, lsv_j) | (lsv_m \prec_t lsv_j) \bigwedge (T_j \overset{rw}{\dashrightarrow} T_i) \bigwedge T_j \in \mathcal{T}_{l,p}\}$;
6:  if $\{\mathcal{D}_{f,c} \bigcup D_{t,c}\}$ has a case in $\mathcal{SSI}_{non}$ then
7:      send "abort" to a local database immediately;
8:  else
9:      $\mathcal{T}_{l,p} \leftarrow \mathcal{T}_{l,p} \bigcup [W_i, \mathcal{D}_{f,c}, \mathcal{D}_{f,p}, D_{t,c}, \mathcal{D}_{t,p}, \perp]$;
10:     Rsend $<W_i, \mathcal{D}_{f,c}, \mathcal{D}_{f,p}, D_{t,c}, \mathcal{D}_{t,p}, \perp>$ to successor;

upon Rreceive $<W_i, \mathcal{D}_{f,c}, \mathcal{D}_{f,p}, D_{t,c}, \mathcal{D}_{t,p}, \perp>$ do
1:  if $W_i \notin \mathcal{T}_{l,p} \bigwedge D_{t,c} \neq (\perp,\perp)$ then /* remote writeset */
2:      $\mathcal{T}_p \leftarrow \mathcal{T}_p \bigcup [W_i, \mathcal{D}_{f,c}, \mathcal{D}_{f,p}, D_{t,c}, \mathcal{D}_{t,p}, \perp]$;
3:      for all  $T_k \in \{T_j | (T_j \overset{rw}{\dashrightarrow} T_i) \bigwedge T_j \in \mathcal{T}_{l,c}\}$  do
4:          if  $D_{t,c}.lsv < lsv_k$  then  $D_{t,c} \leftarrow (k,\ lsv_k)$;
5:          if $(k,\ lsv_k) \in \mathcal{D}_{f,c}$ then  /* a cycle is detected */
6:              $D_{t,c} \leftarrow (\perp,\perp)$;   /* $(\perp,\perp)$ means $T_i$ has a cycle */
7:              break;
8:      end for
9:      $\mathcal{D}_{t,p} \leftarrow \mathcal{D}_{t,p} \bigcup \{(j, lsv_j) | (c_j \prec_t c_i) \bigwedge (D_{t,c}.lsv \prec_t lsv_j)$
                    $\bigwedge (T_j \overset{rw}{\dashrightarrow} T_i) \bigwedge T_j \in \mathcal{T}_{l,p}\}$;
10:     update $\mathcal{T}_p$ with $\mathcal{D}_{f,c}$, $\mathcal{D}_{f,p}$, $D_{t,c}$ and $\mathcal{D}_{t,p}$;
11:     Rsend $<W_i, \mathcal{D}_{f,c}, \mathcal{D}_{f,p}, D_{t,c}, \mathcal{D}_{t,p}, \perp>$ to successor;
12: else if $W_i \notin \mathcal{T}_{l,p} \bigwedge D_{t,c} == (\perp,\perp)$ then /* $W_i$ has a cycle */
13:     Rsend $<W_i, \mathcal{D}_{f,c}, \mathcal{D}_{f,p}, D_{t,c}, \mathcal{D}_{t,p}, \perp>$ to successor;
14: else /* $W_i \in \mathcal{T}_{l,p}$; $W_i$ is a local writeset */
15:     if $D_{t,c} == (\perp,\perp)$ then /* $T_i$ has a cycle */
16:         $\mathcal{T}_{l,p} \leftarrow \mathcal{T}_{l,p} - [W_i, \mathcal{D}_{f,c}, \mathcal{D}_{f,p}, D_{t,c}, \mathcal{D}_{t,p}, \perp]$;
17:         send "abort" to a local database for $T_i$; /* early abort */
18:     Rsend $<W_i, \mathcal{D}_{f,c}, \mathcal{D}_{f,p}, D_{t,c}, \mathcal{D}_{t,p}$, "Decide"$>$ to successor;
19:     gdcpDeliver();

upon Rreceive $<W_i, \mathcal{D}_{f,c}, \mathcal{D}_{f,p}, D_{t,c}, \mathcal{D}_{t,p}$, "Decide"$>$ do
1:  if $W_i \in \mathcal{T}_{l,p}$ then /* $W_i$ is a local writeset */
2:      update $T_i \in \mathcal{T}_{l,p}$ with "Decide";
3:  else /* remote writeset */
4:      $\mathcal{T}_p \leftarrow \mathcal{T}_p \bigcup [W_i, \mathcal{D}_{f,c}, \mathcal{D}_{f,p}, D_{t,c}, \mathcal{D}_{t,p}$, "Decide"];
5:      Rsend $<W_i,, \mathcal{D}_{f,c}, \mathcal{D}_{f,p}, D_{t,c}, \mathcal{D}_{t,p}$, "Decide"$>$ to successor;
6:  gdcpDeliver();

procedure gdcpDeliver ()
1:  while $\mathcal{T}_p.first == <W_i, \mathcal{D}_{f,c}, \mathcal{D}_{f,p}, D_{t,c}, \mathcal{D}_{t,p}$, "Decide"$>$ do
2:      if $T_i \in \mathcal{T}_{l,p}$ then /* local writeset */
3:          $\mathcal{T}_{l,p} \leftarrow \mathcal{T}_{l,p} - [W_i, \mathcal{D}_{f,c}, \mathcal{D}_{f,p}, D_{t,c}, \mathcal{D}_{t,p}]$;
4:          if $T_i$ is not involved in a case in $\mathcal{SSI}_{non}$ then
5:              send "commit" to a local database for $T_i$;
6:              $\mathcal{T}_{l,c} \leftarrow \mathcal{T}_{l,c} \bigcup [W_i, \mathcal{D}_{f,c}, \mathcal{D}_{f,p}, D_{t,c}, \mathcal{D}_{t,p}]$;
7:          else
8:              send "abort" to a local database for $T_i$;
9:      else /* remote writeset */
10:         $\mathcal{T}_p \leftarrow \mathcal{T}_p - [W_i, \mathcal{D}_{f,c}, \mathcal{D}_{f,p}, D_{t,c}, \mathcal{D}_{t,p}]$;
11:         if $T_i$ is not involved in a case in $\mathcal{SSI}_{non}$ then
12:             send $W_i$ to a local database to update;
13:             $\mathcal{T}_c \leftarrow \mathcal{T}_c \bigcup [W_i, \mathcal{D}_{f,c}, \mathcal{D}_{f,p}, D_{t,c}, \mathcal{D}_{t,p}]$;
14: end while
```

dency information. If the protocol message contains a cycle mark, a replication manager sends an "abort" message to its local database immediately. After sending the protocol message to a successor, a replication manager calls **gdcpDeliver** to check whether the first writeset in $\mathcal{T}_p$ has the "Decide" mark, so a replication manager can decide for the writeset. The remaining replicas, upon receiving an explicit "Decide" message, can make a decision for remote or local writesets based on aggregated dependency information. We will see how **gdcpDeliver** makes a decision (line 1-6 of the second **Rreceive**).

In **gdcpDeliver**, we consider a writeset in $\mathcal{T}_p$, having the earliest total order. If the first writeset is not marked with "Decide", **gdcpDeliver** should return without processing other writesets queued in $\mathcal{T}_p$ due to the total order delivery. In contrast, if the writeset has the "Decide" mark, we now can decide whether to deliver or discard the writeset based on the global dependency information conveyed in the protocol message. The case we abort the current transaction is when the current transaction is involved in one of non-serializable cases and other involved transactions could be (or were already) delivered earlier than the current transaction. If the decision is made for a local writeset, a replication manager sends a "commit" or "abort" message back to a local database. Otherwise, we send the writeset to a local database for updating. **gdcpDeliver** keeps processing queued writesets having the "Decide" mark likewise until it encounters a writeset without the mark. This ensures the *uniform* total order delivery (line 1-14 of **gdcpDeliver**).

For additional details of GDCP, which explains an execution of the protocol with an example transaction schedule, readers are referred to Appendix B.

5. SYSTEM ARCHITECTURE

We implemented a research prototype that integrates RSSI into Postgres-R, a replicated PostgreSQL server that uses SI as concurrency control, inspired by Wu et al. [27], and that it is maintained as an open source project.

Our system design is following two design principles: firstly, we strive for high overall system performance (in terms of throughput) by an efficient implementation of RSSI. This required some non-trivial engineering efforts to minimize the overhead of RSSI compared to traditional SI-based replication. Secondly, we are interested in a modular system design that would allow for smooth adaptation of RSSI into existing database systems. We hence implement all SSI-related data structures and algorithms inside a dedicated replication manager that is separated from the DBMS. This minimizes the changes needed to the underlying database engine while being readily portable to any SI-based database system. Appendix C gives a detailed discussion of our Postgres-RSSI implementation.

6. PERFORMANCE EVALUATION

In the following, we report on the results of an experimental performance analysis of RSSI, in which we quantify its overhead and scalability behavior as compared to state-of-the-art SI-based replication approaches from the literature.

6.1 Evaluation Setup

The experiments were conducted on a database server cluster with eight nodes, each with a 3.2GHz Intel Pentium IV CPU and 2GB RAM under RedHat Enterprise Linux version 4 (Linux kernel 2.6.9-11), and interconnected with a Gigabit Ethernet network. We compare our design with two others, which use the same overall system architecture (kernel-based replication and log-based writeset handling, taken from the open-source implementation of Postgres-R, with GDCP instead of Spread for total-ordered broadcast) but have different conflict management algorithms. The baseline we call RSI, using *ww*-conflict management for 1-copy SI [27]. The other systems give serializability: RSSI is our system, and CP-ROO aborts every detected *rw*-conflict but does not globally certify read-only transactions (this is the certification approach used by Bornea et al. [6]). All databases are configured with default settings.

6.2 TPC-C Workload

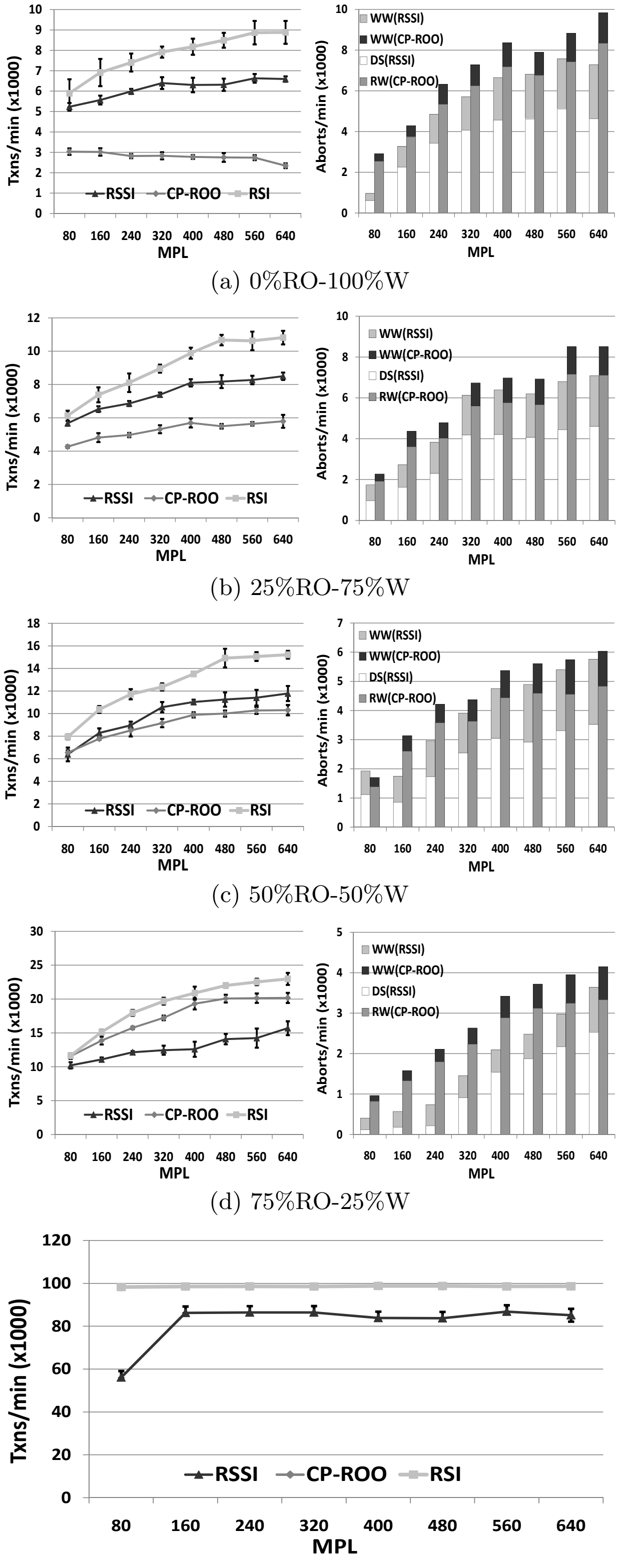
As a starting point, we ran a series of experiments based on the TPC-C++ benchmark[3] in order to evaluate the effectiveness and scalability of the different certification rules. As it turns out, such a realistic workload has so few SI anomalies so that all three approaches achieve the same performance and scalability with no measurable difference, regardless whether they provided 1-copy SI or 1-copy serializability (as does our RSSI). We have placed these results in Appendix D.2.

6.3 Synthetic Workload

Next, we want to focus on the performance impact of our RSSI concurrency control, especially comparing the overhead of RSSI and CP-ROO approaches. We hence created a simple synthetic micro-benchmark called `ssibench` with numerous read-write conflicts between a query transaction and an update transaction, or among update transactions. Appendix D.1.1 gives detailed explanation of `ssibench`.

In the following, we use a fixed number of 8 replicas and we vary the multiprogramming level from 80 to 640, with all clients trying to execute transactions as fast as possible without think time in between. All plotted points in Figure 1 are the average of 5 runs, each run consists of 1 minute ramp-up period and 1 minute measurement period. We vary the portion of read-only transactions from 0% up to 100% to explore performance characteristics for RSSI and CP-ROO. In RSSI, all transactions can be involved in a cycle so that any of them can be a victim when non-serializable cases are detected. Meanwhile, CP-ROO is only concerned with *rw*-edges among update and insert transactions. Obviously, RSSI has an advantage of detecting non-serializable cases in a finer-grained way while it has the overhead of checking upon all types of transactions. In contrast, CP-ROO has a merit of allowing all read-only transactions to commit without certification, but it has a shortcoming of seeing only a single *rw*-edge, instead of multiple edges. Our experimental results show a performance spectrum that captures representative characteristics of the two concurrency control algorithms. Note that in all experiments, except the 100% read-only workload, RSI allows executions that are *not* serializable.

Figure 1-(a) shows results under 100% write-intensive workload, where all transactions in all concurrency control algorithms have to pass through GDCP for certification. In this scenario, we could measure serializable concurrency control overhead and compare false positive abort rate between RSSI and CP-ROO. Throughput graphs of RSI and RSSI slowly increase as MPL grows, whereas CP-ROO is already



(a) 0%RO-100%W

(b) 25%RO-75%W

(c) 50%RO-50%W

(d) 75%RO-25%W

(e) 100%RO-0%W

Figure 1: Throughput and abort rates of `ssibench` with three tables, each has 100K items, at RSI, RSSI and CP-ROO. Abort causes are denoted as WW (*ww*-conflict), DS (*descending structure*) and RW (*read impact*). RO and W stand for read-only and write transactions.

[3] TPC-C++ has the same schema as TPC-C but adds a new transaction type *Credit Check* to induce SI-anomalies [7].

saturated and decreased slowly. When MPL is 80, the abort rate of CP-ROO is three times higher than RSSI; most aborts are due to *rw*-conflicts (i.e., *read impact* in [6]). As MPL increases up to 640, we can observe the widening performance gap between RSSI and CP-ROO; CP-ROO still has very high abort rate at MPL=640, while RSSI could lessen the effect of false positive abort since it checks presence of the *descending structure*. In a write-intensive workload, which has cycles and all types of conflicts, the finer-grained dependency checking of RSSI could be much better than the coarser-grained checking of CP-ROO.

In Figure 1-(b), (c) and (d), we run experiments under workloads that have varying proportions of read-only transactions. When read-only transactions are involved, we can see the trade-off of merits and shortcomings of two methods; in particular, as the portion of read-only transactions increases, we could see a cross-over point where CP-ROO starts to outperform RSSI. With 25% read-only, Figure 1-(b) shows that CP-ROO narrows the performance gap with RSSI by committing read-only transactions immediately. The throughput of CP-ROO also scales slowly as MPL grows. As we increase the portion of read-only transactions up to 50% (i.e., Figure 1-(c)), both RSSI and CP-ROO deliver almost the same throughput although CP-ROO still has a higher abort rate than RSSI. Here the performance gain obtained by the advantage of RSSI balances the overhead of the algorithm. Figure 1-(d) shows that the optimization of CP-ROO, that avoids checks for read-only transactions, is beneficial when the workload is dominated by read-only transactions; here CP-ROO outperforms RSSI whose certification overhead for read-only transactions is evident. We note that CP-ROO still has a higher abort rate for write transactions than RSSI.

In Figure 1-(e), we measured the throughput when the workload consists of only read-only transactions, where all algorithms are serializable under this scenario. In this experiments, RSI and CP-ROO show almost the same performance since no certification is needed, but RSSI still has the certification overhead and shows the overhead in the figure accordingly. The large performance gap when MPL is small (i.e., 80) is because queries and readsets are not fully pipelined yet in this stage. As MPL grows, readsets fully saturate the GDCP network and this narrows the performance gap with other two algorithms.

Appendix D.1.2 provides the breakdown of GDCP overhead under the same synthetic workloads.

7. CONCLUSION

This paper presented a new approach to global serializable execution over replicated databases using SI. We introduced (and proved) a clear criterion that is sufficient for 1-copy serializability, based on absence of *ww*-conflicts and on absence of certain patterns of two consecutive *rw*-conflicts. Based on this theorem, we developed our RSSI algorithm that allows to efficiently check for the absence of those transactional patterns in replicated databases. We incorporated our approach into the Postgres-R(SI) system. RSSI was shown to perform very well in a variety of cases with high update rates, by avoiding unnecessary aborts from false positives as found in the CP-ROO approach that prevents single *rw*-edges. For a TPC-C workload, RSSI can achieve 1-copy serializability for replicated databases with no performance degradation.

8. REFERENCES

[1] A. Adya. *Weak consistency: a generalized theory and optimistic implementations for distributed transactions*. PhD thesis, MIT Lab for Computer Science, 1999.

[2] M. Alomari, A. Fekete, and U. Röhm. A robust technique to ensure serializable executions with snapshot isolation DBMS. In *Proccedings of ICDE 2009*, pages 341–352, 2009.

[3] Y. Amir, C. Danilov, M. Miskin-Amir, J. Schultz, and J. Stanton. The Spread Toolkit: Architecture and Performance. Technical Report CNDS-2004-1, Johns Hopkins University, April 2004.

[4] B. Ban. JGroup – a Toolkit for Reliable Multicast Communication. http://www.jgroup.org, 2007.

[5] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A critique of ANSI SQL isolation levels. In *Proceedings of SIGMOD 1995*, pages 1–10, 1995.

[6] M. A. Bornea, O. Hodson, S. Elnikety, and A. Fekete. One-copy serializability with snapshot isolation under the hood. In *Proceedings of ICDE 2011*, pages 625–636, 2011.

[7] M. J. Cahill, U. Röhm, and A. Fekete. Serializable isolation for snapshot databases. *ACM TODS*, 34(4):1–42, 2009.

[8] M. Castro and B. Liskov. Practical Byzantine fault tolerance and proactive recovery. *ACM TOCS*, 20(4):398–461, 2002.

[9] E. Cecchet, G. Candea, and A. Ailamaki. Middleware-based database replication: the gaps between theory and practice. In *Proceedings of SIGMOD 2008*, pages 739–752, 2008.

[10] K. Daudjee and K. Salem. Lazy database replication with snapshot isolation. In *Proc. of VLDB06*, pages 715–726, 2006.

[11] S. Elnikety, S. Dropsho, and F. Pedone. Tashkent: uniting durability with transaction ordering for high-performance scalable database replication. In *EuroSys*, pages 117–130, 2006.

[12] S. Elnikety, S. Dropsho, and W. Zwaenepoel. Tashkent+: memory-aware load balancing and update filtering in replicated databases. In *Proc. of EuroSys 2007*, pages 399–412, 2007.

[13] S. Elnikety, W. Zwaenepoel, and F. Pedone. Database replication using generalized snapshot isolation. In *Proceedings of SRDS 2005*, pages 73–84, 2005.

[14] A. Fekete, D. Liarokapis, E. O'Neil, P. O'Neil, and D. Shasha. Making snapshot isolation serializable. *ACM TODS*, 30(2):492–528, 2005.

[15] J. Gray, P. Helland, P. O'Neil, and D. Shasha. The dangers of replication and a solution. In *SIGMOD*, pages 173–182, 1996.

[16] R. Guerraoui, R. R. Levy, B. Pochon, and V. Quéma. Throughput optimal total order broadcast for cluster environments. *ACM TOCS*, 28(2):1–32, 2010.

[17] B. Kemme and G. Alonso. Don't be lazy, be consistent: Postgres-R, a new way to implement database replication. In *Proceedings of VLDB 2000*, pages 134–143, 2000.

[18] B. Kemme and G. Alonso. A new approach to developing and implementing eager database replication protocols. *ACM TODS*, 25(3):333–379, 2000.

[19] B. Kemme and G. Alonso. Database replication: a tale of research across communities. *PVLDB*, 3(1):5–12, 2010.

[20] B. Kemme, R. Jiménez-Peris, and M. Patiño Martínez. *Database Replication*. Morgan and Claypool, 2010.

[21] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong. Zyzzyva: Speculative Byzantine fault tolerance. *ACM TOCS*, 27(4):1–39, 2009.

[22] L. Lamport, R. Shostak, and M. Pease. The Byzantine generals problem. *ACM TOPLAS*, 4(3):382–401, 1982.

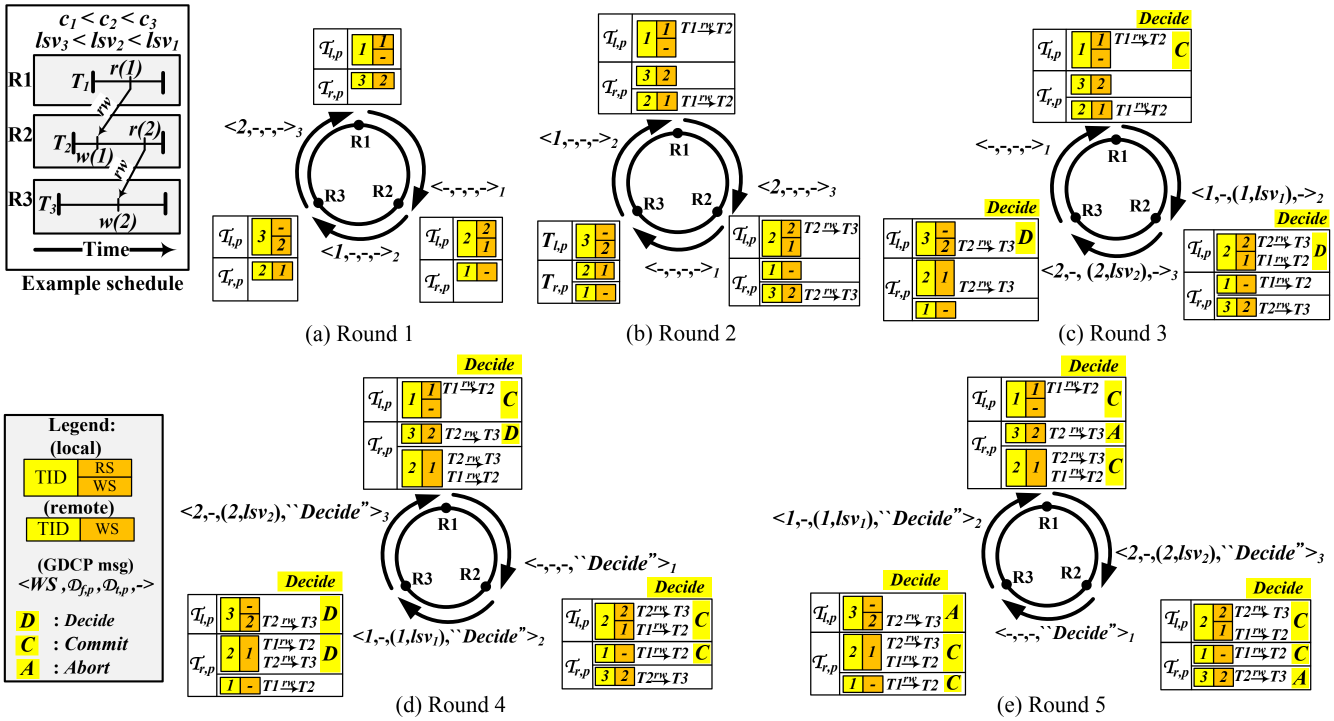
[23] Y. Lin, B. Kemme, M. Patiño Martínez, and R. Jiménez-Peris. Middleware based data replication providing snapshot isolation. In *Proceedings of SIGMOD 2005*, pages 419–430, 2005.

[24] M. Patiño Martínez, R. Jiménez-Peris, B. Kemme, and G. Alonso. MIDDLE-R: Consistent database replication at the middleware level. *ACM TOCS*, 23(4):375–423, 2005.

[25] C. Plattner, G. Alonso, and M. T. Özsu. Extending DBMSs with satellite databases. *VLDB Journal*, 17(4):657–682, 2008.

[26] G. Weikum and G. Vossen. *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery*. Morgan Kaufmann, 2001.

[27] S. Wu and B. Kemme. Postgres-R(SI): Combining replica control with concurrency control based on snapshot isolation. In *Proceedings of ICDE 2005*, pages 422–433, 2005.



(a) Round 1 (b) Round 2 (c) Round 3 (d) Round 4 (e) Round 5

Figure 2: Global dependency checking protocol (GDCP) with an example schedule.

APPENDIX

A. PROOF OF THEOREM 3.1

In order to prove Theorem 3.1, we first establish a lemma about the relationships of the various timestamps along the three types of edges.

Lemma A.1. *Let h be a history over a set of transactions, such that one can assign to each transaction T_i an event $snapshot_i$ preceding b_i and obeying the two conditions reads-from-snapshot and first-committer-wins. Suppose that h contains both c_i and c_j, and that $SG(h)$ contains an edge $T_i \longrightarrow T_j$. Then we have*

- *If $T_i \xrightarrow{wr} T_j$, then $commit(T_i) \preceq lsv(T_j)$*
- *If $T_i \xrightarrow{ww} T_j$, then $commit(T_i) \preceq lsv(T_j)$*
- *If $T_i \xrightarrow{rw} T_j$, then $begin(T_i) \prec commit(T_j)$.*

Proof. If $T_i \xrightarrow{wr} T_j$, then h contains the event $R_j(X_i)$ for some item X. That is, X_i is one of the versions read by T_j, and so T_i is among the transactions whose commit-timestamps are considered in forming $lsv(T_j)$. Since lsv is chosen as the largest timestamp among those considered, we have $commit(T_i) \preceq lsv(T_j)$.

If $T_i \xrightarrow{ww} T_j$, then there is an item X such that h contains $W_i(X_i)$ and $W_j(X_j)$, and the version order places X_i before X_j. That is, c_i precedes c_j. By first-committer-wins, c_i precedes $snapshot_j$. Thus X_i is one of the versions accessed by T_j, and so T_i is among the transactions whose commit-timestamps are considered in forming $lsv(T_j)$. Since lsv is chosen as the largest timestamp among those considered, we have $commit(T_i) \preceq lsv(T_j)$.

If $T_i \xrightarrow{rw} T_j$, then there is an item X such that T_i reads a version X_k which is earlier in the version order than X_j written by T_j. That is, T_i does not see the version X_j. By condition read-from-snapshot, we see that $snapshot_i$ precedes c_j. That is, in terms of timestamps, $begin(T_i) \prec commit(T_j)$. □

Proof Of Theorem 3.1. For contradiction, suppose that h is not 1-copy serializable. Thus there must exist a cycle among the committed transactions in the serialization graph $SG(h)$. Consider in this cycle, the transaction T_p with the highest value for $lsv(T_p)$. In traversing the cycle, consider the two transactions visited immediately after T_p, call these T_f and T_t. That is, the cycle contains edges $T_p \longrightarrow T_f \longrightarrow T_t$.

Now, the choice of T_p as having the largest lsv means that $lsv(T_f) \preceq lsv(T_p)$, and $lsv(T_t) \preceq lsv(T_p)$. We claim that each of the two edges following from T_p is a rw-edge (and thus the pair of edges forms a descending structure, contradicting the hypothesis that the graph has no descending structure).

All that remains is to prove the claims. To prove that $T_p \xrightarrow{rw} T_f$, we show that the edge cannot be ww nor wr. If either were the case, Lemma A.1 would give $commit(T_p) \preceq lsv(T_f)$, and since $lsv(T_p) \prec commit(T_p)$, we would deduce $lsv(T_p) \prec lsv(T_f)$ contradicting the choice of T_p. Now, knowing that $T_p \xrightarrow{rw} T_f$, we use the third part of Lemma A.1 to see that $begin(T_p) \prec commit(T_f)$, and therefore $lsv(T_p) \prec commit(T_f)$. To prove that $T_f \xrightarrow{rw} T_t$, we show that the edge cannot be ww nor wr. If either were the case, Lemma A.1 would give $commit(T_f) \preceq lsv(T_t)$, and we could combine this with the above fact $lsv(T_p) \prec commit(T_f)$, to show $lsv(T_p) \prec lsv(T_t)$, contradicting the choice of T_p to have maximal lsv in the cycle. This completes the proof of the claims, and thus the proof of the theorem. □

B. GDCP DETAILS

B.1 Execution of GDCP

In order to better understand the design of GDCP let us explain the execution of GDCP with an example schedule shown in Figure 2. Suppose that there are three replicas R_1, R_2 and R_3 with their own local transactions T_1, T_2 and T_3, where $lsv_3 \prec_t lsv_2 \prec_t lsv_1$ and $c_1 \prec_t c_2 \prec_t c_3$. The three transactions are concurrent and ready to commit. To simplify the example, we assume the absence of committed transactions in all replicas, so our focus is on four fields in a protocol message; $<WS$, $\mathcal{D}_{f,p}$, $\mathcal{D}_{t,p}$, msg$>$. For convenience, $\mathcal{T}_{r,p}$ denotes a transactional log for remote pending transactions, like what $\mathcal{T}_{l,p}$ represents local pending transactions.

At the onset of commit, three replicas have information for their local transaction. In round 1, all replicas create and send their protocol message to their successor, and at the same time receive a message from their predecessor. After updating $\mathcal{T}_{r,p}$, replicas check an anti-dependency from a local readset to the received writeset, but cannot find any in this round (Figure 2-(a)). In round 2, all writesets convey no information and are forwarded to proper successors. A replication manager performs dependency checking over a newly received writeset and updates $\mathcal{T}_{r,p}$ for the received remote writeset. Replicas R_1 and R_2 detect anti-dependencies and store the anti-dependency information in $\mathcal{D}_{t,p}$ of the corresponding protocol messages (Figure 2-(b)).

In round 3, upon receiving their local protocol message, all replicas acknowledge the end of the first circulation of the local writeset in the ring topology, so they need start the second circulation of the protocol message with "Decide". Then all replicas call **gcdpDeliver** to check the possibility of delivering their local writeset. Among replicas, only R_1 can deliver the local writeset since T_1 has the earliest total order among three transactions. The other two replicas only mark "Decide" in their local writeset and wait until all other remote writesets would be checked in total order (Figure 2-(c)).

In round 4, all writesets circulate the ring topology with "Decide" and updated dependency information. Upon receiving these writesets, R_2 can safely deliver two writesets for T_1 and T_2 in total order, whereas R_1 and R_3 have to wait another round for the protocol messages for T_2 and T_1, respectively (Figure 2-(d)). In round 5, all replicas obtain complete dependency information and uniformly decide which writeset has to be discarded and which one needs to be delivered; in particular, the writeset for T_3 has to be discarded because it is involved in a descending structure of T_2 and T_3 is the last to commit (Figure 2-(e)).

As we learn from the example, all participating replicas uniformly decide for all writesets based on the same global dependency information conveyed in circulated protocol messages, strictly abiding by the invariant of GDCP. We omit a correctness proof for GDCP in this paper due to the space limitation.

C. IMPLEMENTATION DETAILS OF RSSI

C.1 RSSI Replication Manager

The key component of our RSSI architecture in Figure 3 is the *Replication Manager* whose main purpose is to run GDCP. It interacts with the local database to receive (or deliver) writesets either for consistent database update or for global dependency checking. The replication manager exchanges messages with the local database instance in two cases: first, when the local database is ready to commit, it transfers both a readset and a writeset of a transaction to the replication manager. Upon the receipt of these tuple sets, the replication manager sends the writeset using GDCP to the other replicas while keeping the readset for dependency checking. The second case is when the replication manager delivers to the local database a serializable writeset for updating. The existing PostgreSQL components do not need to maintain any dependency information, rather all the information is kept in the replication manager. For efficient event handling, the replication manager is multi-threaded, and the communication to/from the local database is using shared memory communication.

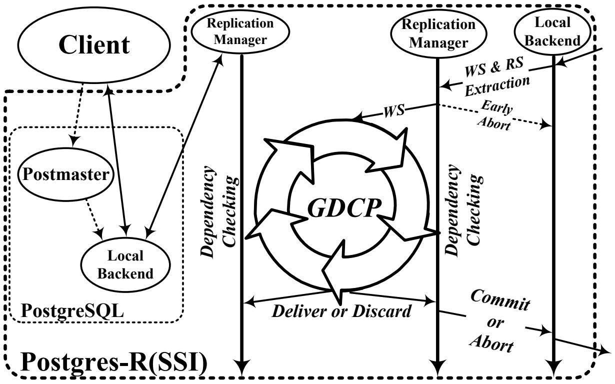
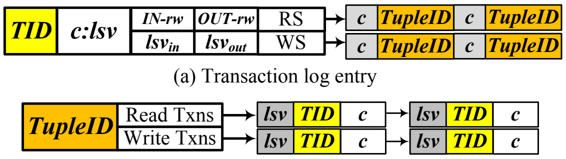


Figure 3: The architecture of Postgres-RSSI.

C.1.1 Data Structures

Postgres-RSSI's replication manager maintains two data structures for global dependency checking of each transaction: a *transaction log* and a *tuple lookup table*. To make these data structures succinct and efficient, we represent both tuples and transactions with fixed-sized formats; we use an integer format for a transaction ID and 160 bits of SHA-1 (or similar) hash digest of (*relation name*, *primary key*) for an unique tuple ID.



(a) Transaction log entry

(b) Tuple lookup table entry

Figure 4: Transaction log and lookup table entries.

The *transaction log* has two roles: firstly, it provides anti-dependency information for each committed/pending transaction; secondly, it is used for garbage-collecting entries in the tuple lookup table (discussed in Appendix C.2.3). As shown in Figure 4-(a), each log entry contains information about the presence of incoming and outgoing anti-dependencies (*IN-rw*, *OUT-rw*) and the corresponding latest-snapshot-version timestamps for those anti-dependencies (lsv_{in}, lsv_{out}), as well as the transaction's readset (RS) and writeset (WS). Log entries are updated either when

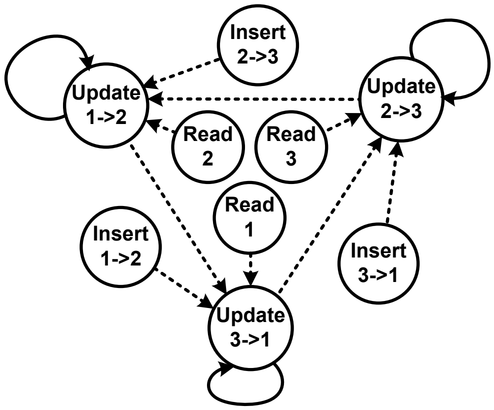


Figure 5: Static dependency graph for `ssibench`, with *rw*-conflicts shown as dashed edges and *ww*-conflicts shown in bold. There is a *wr*-edge not shown that revereses each *rw*-edge. Read i means that a read-only transaction reads items from `ssibench-i`. Update/insert transactions are denoted as $name\ i \rightarrow j$: a $name$ transaction reads from `ssibench-i` and updates/inserts rows in `ssibench-j`.

the replication manager receives the dependency information conveyed in $\mathcal{D}_{f,c}$ and $\mathcal{D}_{f,p}$ of remote transaction's protocol message, or when the local replication manager updates the log by its own local transaction during GDCP.

The replication manager also maintains a *tuple lookup table* to efficiently detect anti-dependencies from/to a given readset/writeset. This lookup table is organised as an inverted index: each tuple entry points to a linked list of pairs of transaction IDs and the corresponding snapshot version (lsv), which read or updated that tuple. To further improve the lookup performance, the TID list is sorted by lsv, and the tuple lookup table itself is organised as a hash-based index table. The entry format of the tuple lookup table is shown in Figure 4-(b).

C.1.2 Failure and Recovery

In this paper, we concentrate on giving an overview how RSSI can recover from simple stop-failures. We expect that more complex failure models such as Byzantine Fault Tolerance (BFT) [22] can be dealt with by adapting one of the existing solutions [8, 21]. If a stop-failure happens at any replica, RSSI first shuts down the group and then recovers the replicated databases from any data inconsistency. Inside a single database instance, we rely on the recovery mechanism of the underlying database for data consistency, while consistent recovery across replicas is guaranteed by copying data from a master replica that has the most up-to-date database among all replicas. After setting up a new ring topology, the replicas then initiate operation again.

C.2 Implementation Details

In the following, we take a closer look at some important implementation details of our Postgres-RSSI prototype.

C.2.1 Writeset Handling

There are two common ways to represent and handle writesets: trigger-based replication systems capture logical SQL statements, while log-sniffer based approaches ship some form of log entries. RSSI follows the later approach by representing transaction writesets as a collection of tuple logs of PostgresSQL. This approach has several advantages over the statement-based writeset handling: firstly, log-based writeset handling is typically faster than statement-level writesets as it bypasses any SQL handling routines; but in particular, it guarantees deterministic updating while statement-based methods fail in cases where the original query contains an update operation with non-deterministic value [9].

C.2.2 Resolving Conflicts

An important issue in replicated databases is resolving conflicts either between local and remote transactions, or solely among remote writesets. Note that this issue is independent of the method for handling writesets. When a conflict occurs between remote and local transactions, RSSI gives higher priority to remote writesets over local ones because they passed already global dependency checking; if a local transaction already has an exclusive lock on a tuple that a remote writeset needs to update, the remote transaction forces a local transaction to immediately abort. When conflicts occur among remote writesets, our solution is to serialize the execution for remote writesets by their total order, so no conflicting writesets can be executed concurrently. We only allow remote writesets to run in parallel if their writesets do not overlap with each other. Although parallel updates allow different update ordering for remote writesets, appearing a violation of the uniform total order guarantee, a mapping table from a local TID to a global commit order preserves the correctness.

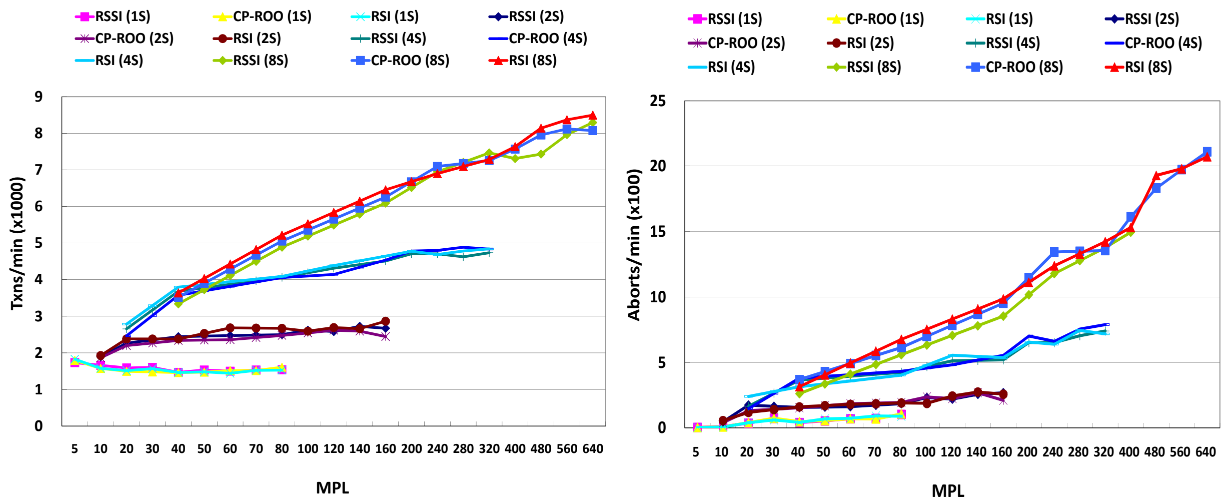
C.2.3 Collecting Garbage Entries

Since the replication manager's data structures reside in memory for efficiency, we must garbage-collect entries in both the transaction log and the tuple lookup table. For cleaning up obsolete entries, we need a staleness criterion. The staleness criterion indicates the entry point before which we could recycle all preceding log entries. In a single system, the log entry for a transaction T_i can be safely removed when there is no pending transaction that starts before T_i commits. In distributed systems, all replication managers have different staleness values.

We avoid running consensus on a staleness point. Each replica has a bookkeeping variable to trace a staleness value, so a replica sends its staleness value to all members whenever the staleness value is updated. The staleness value is an earliest *begin* timestamp of pending transactions. Upon receiving a new staleness value from others, a replication manager takes the oldest of all those received from replicas, and allows clean up of all preceding log entries before this oldest point. When removing log entries, the replication manager can efficiently clean up table entries by checking the readset and writeset arrays of a log entry.

C.3 Limitation

Although our theory can be applied to any system, the architecture presented here may not perform well if replicas are far apart (e.g., on a WAN). The high cost of propagating updates in long delay networks is the main performance bottleneck. Such an *embarrassingly distributed* database might need completely different architectures and algorithms. In future we will explore a practical solution for data management services in cloud platforms, where consistency across data centers is a concern.



(a) Throughput

(b) Abort rate

Figure 6: Throughput and abort rates of TPC-C++ with varying number of replicas and MPL for RSI, RSSI and CP-ROO.

D. EXPERIMENTAL DETAILS

D.1 Synthetic Workload

D.1.1 Synthetic Micro-benchmark

The synthetic micro-benchmark uses three tables called `ssibench-{1, 2, 3}` with one non-null integer and ten variable sized character columns; the integer value is a primary key. Each of the three tables is populated with randomly chosen 100K items. To measure the serializable concurrency control overhead, we use three types of queries: a query transaction (read-only), an update transaction and an insert transaction. All three transactions read a predefined number of consecutive rows starting at a randomly selected row. An update(or insert) transaction updates (or inserts) a predefined number of rows chosen from a uniform random distribution of the identifiers. To create cycles, we configured update/insert transactions to have the following access rules; if a transaction reads items from `ssibench-i`, then update/insert rows from/to `ssibench-j`, where $j{=}i+1 \texttt{ mod } 3$. The static dependency graph for `ssibench` is depicted in Figure 5.

D.1.2 GDCP Overhead

To provide insight on the GDCP behavior in the RSSI architecture, we measured average elapsed time spent during the execution of a successful transaction, compared to the sum of time spent in the GDCP code at the different nodes. Due to space limitation, we present the results with MPL=560 (high load condition) and 8 replicas under `ssibench`. Table 1 shows this breakdown. The relative ratio of GDCP overhead to the lifetime of a transaction is between 13% and 30%, except the case where a workload consists of 100% read-only transactions. The interesting result is that if a workload contains write transactions, the lifetime of read-only transactions is significantly elongated, as the portion of write transactions increases. We conjecture this may be due to an increased amount of dependency information for read-only transactions, but more investigation is needed.

Txn Type / Workload	Read Only		Read-Insert		Read-Update	
	Txn Time	GDCP Time	Txn Time	GDCP Time	Txn Time	GDCP Time
R:W=0:100	N/A	N/A	2397 ms	408 ms (17%)	3853 ms	514 ms (13%)
R:W=25:75	2335 ms	718 ms (30%)	2619 ms	709 ms (27%)	3915 ms	807 ms (20%)
R:W=50:50	2151 ms	555 ms (25%)	2397 ms	611 ms (25%)	3740 ms	650 ms (17%)
R:W=75:25	1791 ms	407 ms (22%)	2028 ms	411 ms (20%)	2019 ms	466 ms (23%)
R:W=100:0	500 ms	379 ms (75%)	N/A	N/A	N/A	N/A

Table 1: The breakdown of transaction execution time measured under synthetic workloads. Numbers in parenthesis give the relative ratio of time spent during GDCP operations to the entire lifetime of a transaction.

D.2 The TPC-C++ Benchmark

We also ran a series of experiments based on a more traditional database benchmark. We are interested in the scalability of our RSSI approach with increasing cluster size. Since Fekete et al. [14] formally proved that TPC-C is serializable when run at snapshot isolation, Cahill et al. [7] proposed TPC-C++ in order to evaluate the effectiveness and overhead of avoiding SI anomalies. We use TPC-C++ and conduct experiments. We run experiments from 1 server to 8 servers by varying MPL from 5 to 640. Plotted points are also the average of 5 runs with 1 minute ramp-up period and 1 minute measurement period.

Figure 6 shows the throughput and abort rate for three algorithms. Note that TPC-C++ is not designed to create numerous cycles as we have in `ssibench`. The breakdown of aborts is not presented in this paper, but is similar to what is found in [7]. We observe the performance scalability under an update-intensive workload with very low probability of having non-serializable cases. As the number of servers increases, the throughput behavior of all three algorithms is the same as we could expect. As we can see in Figure 6(b), most of aborts are *ww*-conflicts, not non-serializable errors.