Analysing and Optimising Parallel Snapshot Isolation

Giovanni Bernardi[1], Andrea Cerone[1], Alexey Gotsman[1], and Hongseok Yang[2]

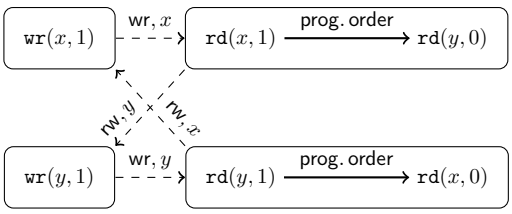
[1]IMDEA Software Institute [2] University of Oxford

Introduction. To achieve availability and scalability, modern Internet services often rely on *geo-replicated databases*, which maintain multiple replicas of data in geographically distinct locations. The database clients can execute transactions on the data at any of the replicas, and these replicas communicate changes to each other using message passing. Ideally, we want the database to provide strong guarantees about transaction processing, such as serialisability. Unfortunately, achieving this requires excessive synchronisation among replicas, which increases latency and limits scalability. This has motivated a recent exploration of consistency models for transactions that weaken the consistency guarantees in exchange for gains in availability and scalability (e.g., [8, 6, 2]). The models being proposed for geo-replicated databases differ in how much they weaken consistency and, in particular, which non-serialisable behaviours (*anomalies*) they expose to application programmers. Unfortunately, we currently lack a systematic understanding of when programmers can use a particular weaker consistency model without violating correctness. And when an application is correct on a given consistency model, we do not know whether the model can safely be weakened even further to improve performance.

We report on a work in progress to address these issues. As a first step, we focus on just one consistency model—Parallel Snapshot Isolation (PSI) [8, 6], which weakens the classical snapshot isolation [3] in a way that allows more scalable geo-replicated implementations. We propose two *static analysis* techniques for applications using a PSI database. First, we present a criterion for checking if an application is *robust*, i.e., behaves the same whether using a PSI database or a serialisable one. This follows up on an existing robustness criterion for the classical snapshot isolation [5], which has proved useful for understanding when it is safe to use this consistency level. Second, we propose a criterion for checking when transactions running on PSI can be *chopped* into smaller pieces without introducing new behaviours [7]. Chopping transactions can improve performance [9]. Thus, when PSI is acceptable for an application, our criterion enables programmers to optimise the code of their transactions to improve scalability even further. The proofs of soundness of our static analyses exploit a novel axiomatic formulation of PSI that we describe in a companion submission [4].

Our results are initial steps towards understanding how the behaviour of applications is affected by consistency models of geo-replicated databases. We hope that further work in this direction will allow us to come up with consistency models that strike an optimal trade-off between achieving scalability and providing meaningful guarantees to applications.

Parallel snapshot isolation. A PSI database consists of a number of replicas, each storing all objects in the database. A transaction initially executes at a single replica and reads object values from a snapshot of the replica state. PSI precludes write conflicts: when two concurrent transactions write to the same object, one of them must abort. A transaction first commits at the original replica, after which its effects are propagated asynchronously to other replicas. Unlike snapshot isolation, PSI does not enforce a global ordering on committed transactions: these are propagated between replicas in causal order. For this reason, PSI admits executions that are not allowed by snapshot isolation (and, consequently, serialisability). One of such executions is a *long fork* anomaly shown in the figure (boxes denote transactions; edges between them are explained in the following). We

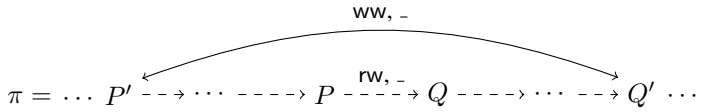


have two concurrent transactions writing to objects x and y. A third transaction sees the write to x, but not y, and a fourth one sees the write to y, but not x. This anomaly can happen when each transaction executes at a separate replica, and the messages about the writes to x and y are delivered to the replicas executing the reading transactions in different orders.

Robustness. We assume a set $\mathcal{P} = \{P_1, \ldots, P_n\}$ of *transactional programs*, defining the code of transactions in an application. At run-time, an application can call each program multiple times, thus giving rise to multiple transactions. We do not fix a particular programming language, but assume that, for each $P_i \in \mathcal{P}$, we are given the sets of the objects that *may* be read or written by a transaction arising from executing P_i (sets R_i^+ and W_i^+), or *must* be written by every transaction arising from P_i (set W_i^-); in particular $W_i^- \subseteq W_i^+$. Based on these sets, we construct a *static dependency graph* $\mathsf{SDG}(\mathcal{P}) = (\mathcal{P}, \dashrightarrow, \longleftrightarrow)$. Its nodes are transactional programs, and its edges describe possible *dependencies* between different programs, analogous to those in serialisation graphs [1]. The directed edges $\dashrightarrow$ describe the dependencies that may exist at run-time, and the undirected edges $\longleftrightarrow$ the dependencies that must always exist. We define the edges between programs P_i and P_j as follows:

- $P_i \overset{\mathsf{wr},x}{\dashrightarrow} P_j$, if $x \in W_i^+ \cap R_j^+$: P_j may read the value of an object x written by P_i;
- $P_i \overset{\mathsf{rw},x}{\dashrightarrow} P_j$, if $x \in R_i^+ \cap W_j^+$: P_j may overwrite the value of an object x read by P_i;
- $P_i \overset{\mathsf{ww},x}{\dashrightarrow} P_j$, if $x \in W_i^+ \cap W_j^+$: P_j may overwrite the value of an object x written by P_i;
- $P_i \overset{\mathsf{ww},x}{\longleftrightarrow} P_j$, if $x \in W_i^- \cap W_j^-$: P_i must overwrite the value of an object x written by P_i or vice versa.

The following notion of necessity generalises that of vulnerability used in the robustness criterion for classical snapshot isolation [5]. In a path π in the graph $\mathsf{SDG}(\mathcal{P})$, an edge $P \overset{\mathsf{rw},\_}{\dashrightarrow} Q$ is *necessary*, if for every P' before P on π (or equal to it), and every Q' after Q on π (or equal to it), $\mathsf{SDG}(\mathcal{P})$ contains no must edge $P' \overset{\mathsf{ww},\_}{\longleftrightarrow} Q'$. For instance, in the following path the edge $P \overset{\mathsf{rw},\_}{\dashrightarrow} Q$ is not necessary:

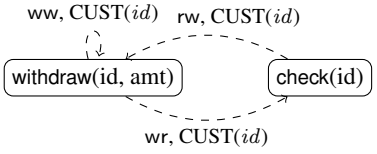


Theorem 1 (Robustness)**.** *Assume* $\mathsf{SDG}(\mathcal{P})$ *has no cycle that contains at least two necessary edges* $P \overset{\mathsf{rw},x}{\dashrightarrow} Q$ *and* $P' \overset{\mathsf{rw},y}{\dashrightarrow} Q'$ *with* $x \neq y$*. Then* $\mathcal{P}$ *behaves the same whether using a PSI database or a serialisable one.*

As shown in the figure with a long fork anomaly, the dependency graph of the corresponding program must contain a critical cycle. We now apply Theorem 1 to prove an application robust. Consider a database with tables CUST and SSN, respectively for customers and social security numbers. The fields of CUST are an identifier, a balance, and a foreign key into the SSN table. The following transactional programs satisfy the conditions of the theorem and, hence, produce only serialisable behaviours:

```
withdraw(id, amount) {
  (bal, _) = CUST(id).read;
  if (bal>amount)
    CUST(id).write(bal-amount);
}
```

```
check(id) {
  (bal, ssn_id)=
    CUST(id).read;
  if (bal<threshold)
    ssn_cust =
      SSN(ssn_id).read;
}
```

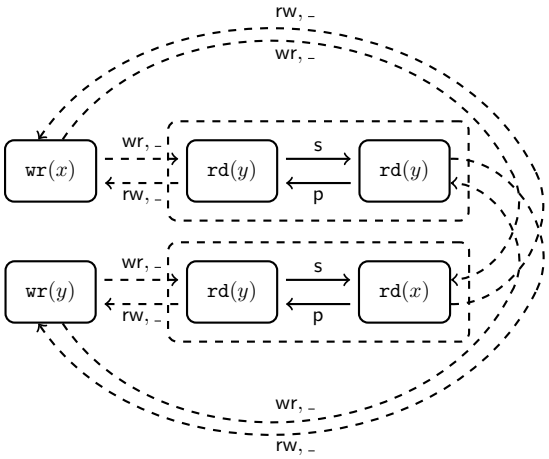


Transaction chopping. We assume a set $\mathcal{P} = \{P_1, \ldots, P_n\}$ of *chain programs*. Each P_i defines the code of a transaction, chopped into a *chain* of *pieces* $Q_{i,1}, \ldots, Q_{i,k_i}$. A program P_i can be executed either as a single *coarse-grained* transaction, or as a sequence of *fine-grained* transactions arising from executing its pieces; the latter is potentially more efficient [9]. The criterion we now define ensures that both ways of executing the programs in $\mathcal{P}$ lead to the same behaviour. As before, we assume that for each piece $Q_{i,j}$ we are given the sets $R^+_{i,j}$ and $W^+_{i,j}$ of objects that it may read or write; the must-write set is not used in this case. The *static chopping graph* $\mathsf{SCG}(\mathcal{P})$ has the set of all pieces $Q_{i,j}$ as vertices and contains the following edges:

- $\xrightarrow{\mathsf{wr},\_}$, $\xrightarrow{\mathsf{rw},\_}$ and $\xrightarrow{\mathsf{ww},\_}$, defined as before;
- $Q_{i,j_1} \xrightarrow{\mathsf{s}} Q_{i,j_2}$, if $j_1 < j_2$: Q_{i,j_2} succeeds Q_{i,j_1} in a chain;
- $Q_{i,j_1} \xrightarrow{\mathsf{p}} Q_{i,j_2}$, if $j_1 > j_2$: Q_{i,j_2} preceeds Q_{i,j_1} in a chain.

Theorem 2 (Chopping)**.** *Assume* $\mathsf{SCG}(\mathcal{P})$ *has no cycle that contains at most one* rw *edge and contains a subpath of the form* $\xrightarrow{\lambda_1}\xrightarrow{\mathsf{p}}\xrightarrow{\lambda_2}$*, where* $\lambda_1, \lambda_2 \in \{(\mathsf{wr}, \_), (\mathsf{ww}, \_), (\mathsf{rw}, \_)\}$*. Then executing the programs in* $\mathcal{P}$ *using a PSI database as coarse-grained transactions or as chains of fined-grained transactions produces the same set of behaviours.*

The theorem gives conditions under which chopping transactions does not introduce new behaviours. These conditions are more permissive than those for transaction chopping under serialisability [7]. As an example, consider the static chopping graph shown below, which could be obtained from an application producing the long fork anomaly. The dashed boxes group pieces into chains. Since the graph has no cycle with at most one rw-edge, we can safely optimise the application by running each of the two read-only transaction as a chain of two smaller transactions.



References

[1] A. Adya. *Weak Consistency: A Generalized Theory and Optimistic Implementations for Distributed Transactions*. Ph.D., MIT, 1999.

[2] P. Bailis, A. Davidson, A. Fekete, A. Ghodsi, J. M. Hellerstein, and I. Stoica. Highly Available Transactions: virtues and limitations. In *VLDB*, 2014.

[3] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A critique of ANSI SQL isolation levels. In *SIGMOD*, 1995.

[4] G. Bernardi, A. Cerone, and A. Gotsman. A uniform formalisation of modern transactional consistency models, 2015. Companion submission.

[5] A. Fekete, D. Liarokapis, E. O'Neil, P. O'Neil, and D. Shasha. Making snapshot isolation serializable. *ACM Trans. Database Syst.*, 30(2), 2005.

[6] M. Saeida Ardekani, P. Sutra, and M. Shapiro. Non-monotonic snapshot isolation: Scalable and strong consistency for geo-replicated transactional systems. In *SRDS*, 2013.

[7] D. Shasha, F. Llirbat, E. Simon, and P. Valduriez. Transaction chopping: Algorithms and performance studies. *ACM Trans. Database Syst.*, 20(3), 1995.

[8] Y. Sovran, R. Power, M. K. Aguilera, and J. Li. Transactional storage for geo-replicated systems. In *SOSP*, 2011.

[9] Y. Zhang, R. Power, S. Zhou, Y. Sovran, M. K. Aguilera, and J. Li. Transaction chains: achieving serializability with low latency in geo-distributed storage systems. In *SOSP*, 2013.