Checking Robustness Against Snapshot Isolation

Sidi Mohamed Beillahi, Ahmed Bouajjani, and Constantin Enea

IRIF, University Paris Diderot & CNRS, {beillahi,abou,cenea}@irif.fr

Abstract. Transactional access to databases is an important abstraction allowing programmers to consider blocks of actions (i.e., transactions) as executing in isolation. The strongest consistency model in this context is serializability, which ensures the atomicity abstraction of transactions executing over a sequentially consistent memory. Since ensuring serializability carries a significant penalty on availability, modern databases provide weaker consistency models, one of the most prominent being snapshot isolation. In general, the correctness of a program relying on serializable transactions may be broken when using weaker models. However, certain programs may also be insensitive to consistency relaxations, which means that all their properties holding under serializability are preserved even when they are executed over some weakly consistent database and without additional synchronization.

In this paper, we address the issue of verifying if a given program is robust against snapshot isolation, i.e., all its behaviors are serializable even if it is executed over a database ensuring snapshot isolation. We show that this verification problem is polynomial time reducible to a state reachability problem in transactional programs over a sequentially consistent shared memory. This reduction opens the door to the reuse of the classic verification technology (under sequential consistency) for reasoning about weakly-consistent programs. In particular, we show that it can be used to derive a proof technique based on Lipton’s reduction theory that allows to prove programs robust.

1 Introduction

Transactions simplify concurrent programming by enabling computations on shared data that are isolated from other concurrent computations and resilient to failures. Modern databases provide transactions in various forms corresponding to different tradeoffs between consistency and availability. The strongest level of consistency is achieved with serializable transactions [20] whose outcome in concurrent executions is the same as if the transactions were executed atomically in some order. Unfortunately, serializability carries a significant penalty on the availability of the system. For this reason, modern databases often provide weaker consistency models, one of the most prominent being snapshot isolation [4]. Then, an important issue is to ensure that the level of consistency

* This work is supported in part by the European Research Council (ERC) under the European Union’s Horizon 2020 research and innovation programme (grant agreement No 678177).
needed by a given application coincides with the one that is guaranteed by its infrastructure, i.e., the database it uses. One way to tackle this issue is to investigate the problem of checking robustness of application programs against relaxations in the consistency guarantees: Given a program $P$ and two consistency models $S$ and $W$ such that $S$ is stronger than $W$, we say that $P$ is robust for $S$ against $W$ if for every two implementations $I_S$ and $I_W$ of $S$ and $W$ respectively, the set of computations of $P$ when running with $I_S$ is the same as its set of computations when running with $I_W$. This means that $P$ is not sensitive to the consistency relaxation from $S$ to $W$, and therefore (1) it is possible to reason about the behaviors of $P$ assuming that it is running over $S$, and (2) no additional synchronization is required when $P$ runs over the weak consistency model $W$ such that it maintains all its properties satisfied with the model $S$.

In this paper, we address the problem of verifying robustness of transactional programs for serializability, against snapshot isolation (SI). Intuitively, under snapshot isolation, any transaction $t$ reads values from a snapshot of the database taken at its start and $t$ is allowed to commit only if no other committed transaction has written to a location that $t$ wrote to, since $t$ started. Robustness is a form of program equivalence between two versions of the same program, obtained using two different semantics, one more permissive than the other one. It ensures that this permissiveness has actually no effect on the particular program under consideration. The difficulty in checking robustness is to apprehend the extra behaviors due to the relaxed consistency model w.r.t. the strong model. This requires a priori reasoning about complex order constraints between operations in arbitrarily long computations, which may need maintaining unbounded ordered structures, and make robustness checking hard or even undecidable.

Our first contribution is to show that verifying robustness of transactional programs against snapshot isolation can be reduced in polynomial time to the reachability problem in concurrent programs under sequential consistency (SC). This allows (1) to avoid explicit handling of the snapshots from where transactions read along computations (since this may imply memorizing an unbounded amount of information), and (2) to leverage available tools for verifying invariants/reachability problems on concurrent programs. Moreover, this implies that the robustness problem is decidable for finite-state programs, PSPACE-complete when the number of sites is fixed, and EXPSPACE-complete otherwise. This is the first result on the decidability and complexity of the problem of verifying robustness in the context of transactional programs. The problem of verifying robustness has been considered in the literature for several consistency models, including eventual and causal consistency \cite{6,8,10,11,19}. These works provide (over- or under-)approximate analyses for checking robustness, but none of them provides precise (sound and complete) algorithmic verification methods for solving this problem, nor addresses its decidability and complexity.

Based on this reduction, our second contribution is a proof methodology for establishing robustness which builds on Lipton’s reduction theory \cite{17}. We use the theory of movers to establish whether the relaxations allowed by SI are harmless, i.e., they don’t introduce new behaviors compared to serializability.
Checking Robustness Against Snapshot Isolation

(a) Write Skew (WS).

\[ p1: \[ r1 = y \quad //0 \mid x = 1 \] \quad \text{conflict} \quad p2: \[ r2 = x \quad //0 \mid y = 1 \] \]

(b) A WS execution trace.

Fig. 1: Examples of non-robust programs illustrating the difference between SI and serializability. causal dependency means that a read in a transaction obtains its value from a write in another transaction. conflict means that a write in a transaction is not visible to a read in another transaction, but it would affect the read value if it were visible. Here, happens-before is the union of the two.

We tested the applicability of the proposed verification techniques on a benchmark suite containing 10 challenging applications extracted from previous work \cite{2, 5, 10, 13, 15, 18, 23}. These techniques were precise enough for proving or disproving the robustness of all of these applications.

2 Overview

In this section, we give an overview of our approach for checking robustness against snapshot isolation. While serializability enforces that transactions are atomic and conflicting transactions, i.e., which read or write to a common location, cannot commit concurrently, SI allows that conflicting transactions commit in parallel as long as they don’t contain a write-write conflict, i.e., write on a common location. Moreover, under SI, each transaction reads from a snapshot of the database taken at its start. These relaxations permit the “anomaly” known as Write Skew (WS) shown in Figure 1a where an anomaly is a program execution which is allowed by SI, but not by serializability. The execution of Write Skew under SI allows the reads of \( x \) and \( y \) to return 0 although this cannot happen under serializability. These values are possible since each transaction is executed locally (starting from the initial snapshot) without observing the writes of the other transaction.

**Execution trace.** Our notion of program robustness is based on an abstract representation of executions called trace. Informally, an execution trace is a set of events, i.e., accesses to shared variables and transaction begin/commit events, along with several standard dependency relations between events recording the data-flow. The transitive closure of the union of all these dependency relations is called happens-before. An execution is an anomaly if the happens-before of its trace is cyclic. Figure 1b shows the happens-before of the Write Skew anomaly. Notice that the happens-before order is cyclic in both cases.

Semantically, every transaction execution involves two main events, the issue and the commit. The issue event corresponds to a sequence of reads and/or writes where the writes are visible only to the current transaction. We interpret it as a single event since a transaction starts with a database snapshot that it updates in isolation, without observing other concurrently executing transactions. The commit event is where the writes are propagated and made visible to all processes. Under serializability, the two events coincide, i.e., they are adjacent
in the execution. Under $\mathcal{SI}$, this is not the case and in between the issue and the commit of the same transaction, we may have issue/commit events from concurrent transactions. When a transaction commit does not occur immediately after its issue, we say that the underlying transaction is delayed. For example, the following execution of WS corresponds to the happens-before cycle in Figure 1b where the write to $x$ was committed after $t_2$ finished, hence, $t_1$ was delayed:

\[
\begin{align*}
\text{begin}(p_1,t_1) & \text{ld}(p_1,t_1,y,0)\text{isu}(p_1,t_1,x,1) & \text{com}(p_1,t_1) \\
\text{begin}(p_2,t_2) & \text{ld}(p_2,t_2,x,0)\text{isu}(p_2,t_2,y,1)\text{com}(p_2,t_2)
\end{align*}
\]

Above, $\text{begin}(p_1,t_1)$ stands for starting a new transaction $t_1$ by process $p_1$, $\text{ld}$ represents read (load) actions, while $\text{isu}$ denotes write actions that are visible only to the current transaction (not yet committed). The writes performed during $t_1$ become visible to all processes once the commit event $\text{com}(p_1,t_1)$ takes place.

**Reducing robustness to SC reachability.** The above $\mathcal{SI}$ execution can be mimicked by an execution of the same program under serializability modulo an instrumentation that simulates the delayed transaction. The local writes in the issue event are simulated by writes to auxiliary registers and the commit event is replaced by copying the values from the auxiliary registers to the shared variables (actually, it is not necessary to simulate the commit event; we include it here for presentation reasons). The auxiliary registers are visible only to the delayed transaction. In order that the execution be an anomaly (i.e., not possible under serializability without the instrumentation) it is required that the issue and the commit events of the delayed transaction are linked by a chain of happens-before dependencies. For instance, the above execution for WS can be simulated by:

\[
\begin{align*}
\text{begin}(p_1,t_1) & \text{ld}(p_1,t_1,y,0)\text{st}(p_1,t_1,x,r_x) & \text{com}(p_1,t_1) \\
\text{begin}(p_2,t_2) & \text{ld}(p_2,t_2,x,0)\text{isu}(p_2,t_2,y,1)\text{com}(p_2,t_2)
\end{align*}
\]

The write to $x$ was delayed by storing the value in the auxiliary register $r_x$, and the happens-before chain exists because the read on $y$ that was done by $t_1$ is conflicting with the write on $y$ from $t_2$ and the read on $x$ by $t_2$ is conflicting with the write of $x$ in the simulation of $t_1$’s commit event. On the other hand, the following execution of Write-Skew without the read on $y$ in $t_1$:

\[
\begin{align*}
\text{begin}(p_1,t_1) & \text{st}(p_1,t_1,x,r_x,1) & \text{com}(p_1,t_1) \\
\text{begin}(p_2,t_2) & \text{ld}(p_2,t_2,x,0)\text{isu}(p_2,t_2,y,1)\text{com}(p_2,t_2)
\end{align*}
\]

misses the conflict (happens-before dependency) between the issue event of $t_1$ and $t_2$. Therefore, the events of $t_2$ can be reordered to the left of $t_1$ and obtain an equivalent execution where $\text{st}(p_1,t_1,x,r_x)$ occurs immediately after $\text{st}(p_1,t_1,x,r_x,1)$. In this case, $t_1$ is not anymore delayed and this execution is possible under serializability (without the instrumentation).

If the number of transactions to be delayed in order to expose an anomaly is unbounded, the instrumentation described above may need an unbounded number of auxiliary registers. This would make the verification problem hard or even undecidable. However, we show that it is actually enough to delay a single transaction, i.e., a program admits an anomaly under $\mathcal{SI}$ iff it admits an anomaly containing a single delayed transaction. This result implies that the number of auxiliary registers needed by the instrumentation is bounded.
Checking Robustness Against Snapshot Isolation 5

by the number of program variables, and that checking robustness against SI can be reduced in linear time to a reachability problem under serializability (the reachability problem encodes the existence of the chain of happens-before dependencies mentioned above). The proof of this reduction relies on a non-trivial characterization of anomalies.

**Proving robustness using commutativity dependency graphs.** Based on the reduction above, we also devise an approximated method for checking robustness based on the concept of mover in Lipton’s reduction theory [17]. An event is a left (resp., right) mover if it commutes to the left (resp., right) of another event (from a different process) while preserving the computation. We use the notion of mover to characterize happens-before dependencies between transactions. Roughly, there exists a happens-before dependency between two transactions in some execution if one doesn’t commute to the left/right of the other one.

We define a commutativity dependency graph which summarizes the happens-before dependencies in all executions of a given program between transactions \( t \) as they appear in the program, transactions \( t \setminus \{w\} \) where the writes of \( t \) are deactivated (i.e., their effects are not visible outside the transaction), and transactions \( t \setminus \{r\} \) where the reads of \( t \) obtain non-deterministic values. The transactions \( t \setminus \{w\} \) are used to simulate issue events of delayed transactions (where writes are not yet visible) while the transactions \( t \setminus \{r\} \) are used to simulate commit events of delayed transactions (which only write to the shared memory). Two transactions \( a \) and \( b \) are linked by an edge iff \( a \) cannot move to the right of \( b \) (or \( b \) cannot move to the left of \( a \)), or if they are related by the program order (i.e., issued in some order in the same process). Then a program is robust if for every transaction \( t \), this graph doesn’t contain a path from \( t \setminus \{w\} \) to \( t \setminus \{r\} \) formed of transactions that don’t write to a variable that \( t \) writes to (the latter condition is enforced by SI since two concurrent transactions cannot commit at the same time when they write to a common variable). For example, Figure 2 shows the commutativity dependency graph of the modified WS program where the read of \( y \) is removed from \( t_1 \). The fact that it doesn’t contain any path like above implies that it is robust.

3 Programs

A program is parallel composition of processes distinguished using a set of identifiers \( \mathbb{P} \). Each process is a sequence of transactions and each transaction is a sequence of labeled instructions. Each transaction starts with a begin instruction and finishes with a commit instruction. Each other instruction is either an assignment to a process-local register from a set \( \mathbb{R} \) or to a shared variable from a set \( \mathbb{V} \), or an assume statement. The read/write assignments use values from a data domain \( \mathbb{D} \). An assignment to a register \( \langle reg \rangle := \langle var \rangle \) is called a read of the shared-variable \( \langle var \rangle \) and an assignment to a shared variable \( \langle var \rangle := \langle reg-expr \rangle \)
is called a write to \(\langle\text{var}\rangle\) (\(\langle\text{reg-expr}\rangle\) is an expression over registers whose syntax we leave unspecified since it is irrelevant for our development). The assume \(\langle\text{bexpr}\rangle\) blocks the process if the Boolean expression \(\langle\text{bexpr}\rangle\) over registers is false. They are used to model conditionals as usual. We use goto statements to model an arbitrary control-flow where the same label can be assigned to multiple instructions and multiple goto statements can direct the control to the same label which allows to mimic imperative constructs like loops and conditionals. In our syntax, we consider simple read/write instructions, however, our results apply as well to instructions that include SQL (select/update) queries. In fact, in our experiments we have programs with SQL based transactions.

The semantics of a program under SI is defined as follows. The shared variables are stored in a central memory and each process keeps a replicated copy of the central memory. A process starts a transaction by discarding its local copy and fetching the values of the shared variables from the central memory. When a process commits a transaction, it merges its local copy of the shared variables with the one stored in the central memory in order to make its updates visible to all processes. During the execution of a transaction, the process stores the writes to shared variables only in its local copy and reads only from its local copy. When a process merges its local copy with the centralized one, it is required that there were no concurrent updates that occurred after the last fetch from the central memory to a shared variable that was updated by the current transaction. Otherwise, the transaction is aborted and its effects discarded.

More precisely, the semantics of a program \(\mathcal{P}\) under SI is defined as a labeled transition system \([\mathcal{P}]_{SI}\) where transactions are labeled by the set of events
\[
\mathcal{E}_v = \{ \text{begin}(p, t), \text{ld}(p, t, x, v), \text{isu}(p, t, x, v), \text{com}(p, t) : p \in \mathcal{P}, t \in T \times T, x \in \mathcal{V}, v \in D \}
\]
where begin and com label transitions corresponding to the start and the commit of a transaction, respectively, isu and ld label transitions corresponding to writing, resp., reading, a shared variable during some transaction.

An execution of program \(\mathcal{P}\), under snapshot isolation, is a sequence of events \(e\mathcal{V}_1 \cdot e\mathcal{V}_2 \cdot \ldots\) corresponding to a run of \([\mathcal{P}]_{SI}\). The set of executions of \(\mathcal{P}\) under SI is denoted by \(\mathcal{E}_{\mathcal{X}}(\mathcal{P})\).

4 Robustness Against SI

A trace abstracts the order in which shared-variables are accessed inside a transaction and the order between transactions accessing different variables. Formally, the trace of an execution \(\rho\) is obtained by (1) replacing each sub-sequence of transitions in \(\rho\) corresponding to the same transaction, but excluding the com transition, with a single “macro-event” isu\((p, t)\), and (2) adding several standard relations between these macro-events isu\((p, t)\) and commit events com\((p, t)\) to record the data-flow in \(\rho\), e.g. which transaction wrote the value read by another transaction. The sequence of isu\((p, t)\) and com\((p, t)\) events obtained in the first step is called a summary of \(\rho\). We say that a transaction \(t\) in \(\rho\) performs an external read of a variable \(x\) if \(\rho\) contains an event ld\((p, t, x, v)\) which is not preceded by a write on \(x\) of \(t\), i.e., an event isu\((p, t, x, v)\). Also, we say that a transaction \(t\) writes a variable \(x\) if \(\rho\) contains an event isu\((p, t, x, v)\), for some \(v\).
The trace $\text{tr}(\rho) = (\tau, \text{PO}, \text{WR}, \text{WW}, \text{RW}, \text{STO})$ of an execution $\rho$ consists of the summary $\tau$ of $\rho$ along with the program order $\text{PO}$, which relates any two issue events $\text{isu}(p, t)$ and $\text{isu}(p, t')$ that occur in this order in $\tau$, write-read relation $\text{WR}$ (also called read-from), which relates any two events $\text{com}(p, t)$ and $\text{isu}(p', t')$ that occur in this order in $\tau$ such that $t'$ performs an external read of $x$, and $\text{com}(p, t)$ is the last event in $\tau$ before $\text{isu}(p', t')$ that writes to $x$ (to mark the variable $x$, we may use $\text{WR}(x)$), the write-write order $\text{WW}$ (also called store-order), which relates any two store events $\text{com}(p, t)$ and $\text{com}(p', t')$ that occur in this order in $\tau$ and write to the same variable $x$ (to mark the variable $x$, we may use $\text{WW}(x)$), the read-write relation $\text{RW}$ (also called conflict), which relates any two events $\text{isu}(p, t)$ and $\text{com}(p', t')$ that occur in this order in $\tau$ such that $t$ reads a value that is overwritten by $t'$, and the same-transaction relation $\text{STO}$, which relates the issue event with the commit event of the same transaction. The read-write relation $\text{RW}$ is formally defined as $\text{RW}(x) = \text{WR}^{-1}(x); \text{WW}(x)$ (we use ; to denote the standard composition of relations) and $\text{RW} = \bigcup_{x \in \mathbb{V}} \text{RW}(x)$. If a transaction $t$ reads the initial value of $x$ then $\text{RW}(x)$ relates $\text{isu}(p, t)$ to $\text{com}(p', t')$ of any other transaction $t'$ which writes to $x$ (i.e., $(\text{isu}(p, t), \text{com}(p', t')) \in \text{RW}(x)$) (note that in the above relations, $p$ and $p'$ might designate the same process).

Since we reason about only one trace at a time, to simplify the writing, we may say that a trace is simply a sequence $\tau$ as above, keeping the relations $\text{PO}$, $\text{WR}$, $\text{WW}$, $\text{RW}$, and $\text{STO}$ implicit. The set of traces of executions of a program $\mathcal{P}$ under $\text{SI}$ is denoted by $\mathbb{T}r(\mathcal{P})_{\text{SI}}$.

**Serializable semantics.** The semantics of a program under serializability can be defined using a transition system where the configurations keep a single shared-variable valuation (accessed by all processes) with the standard interpretation of read and write statements. Each transaction executes in isolation. Alternatively, the serializability semantics can be defined as a restriction of $[\mathcal{P}]_{\text{SI}}$ to the set of executions where each transaction is immediately delivered when it starts, i.e., the start and commit time of transaction coincide $t.s = t.c$. Such executions are called *serializable* and the set of serializable executions of a program $\mathcal{P}$ is denoted by $\mathbb{E}x_{\text{SER}}(\mathcal{P})$. The latter definition is easier to reason about when relating executions under snapshot isolation and serializability, respectively.

**Serializable trace.** A trace $\text{tr}$ is called *serializable* if it is the trace of a serializable execution. Let $\mathbb{T}r_{\text{SER}}(\mathcal{P})$ denote the set of serializable traces. Given a serializable trace $\text{tr} = (\tau, \text{PO}, \text{WR}, \text{WW}, \text{RW}, \text{STO})$ we have that every event $\text{isu}(p, t)$ in $\tau$ is immediately followed by the corresponding $\text{com}(p, t)$ event.

**Happens before order.** Since multiple executions may have the same trace, it is possible that an execution $\rho$ produced by snapshot isolation has a serializable trace $\text{tr}(\rho)$ even though $\text{isu}(p, t)$ events may not be immediately followed by $\text{com}(p, t)$ actions. However, $\rho$ would be equivalent, up to reordering of “independent” (or commutative) transitions, to a serializable execution. To check whether the trace of an execution is serializable, we introduce the happens-before relation on the events of a given trace as the transitive closure of the union of all the relations in the trace, i.e., $\text{HB} = (\text{PO} \cup \text{WW} \cup \text{WR} \cup \text{RW} \cup \text{STO})^+$.
Finally, the happens-before relation between events is extended to transactions as follows: a transaction $t_1$ happens-before another transaction $t_2 \neq t_1$ if the trace $tr$ contains an event of transaction $t_1$ which happens-before an event of $t_2$. The happens-before relation between transactions is denoted by $HB_t$ and called transactional happens-before. The following characterizes serializable traces.

**Theorem 1** ([1, 22]). A trace $tr$ is serializable iff $HB_t$ is acyclic.

A program is called robust if it produces the same set of traces as the serializability semantics.

**Definition 1.** A program $P$ is called robust against $SI$ iff $Tr_{SI}(P) = Tr_{SER}(P)$.

Since $Tr_{SER}(P) \subseteq Tr_{X}(P)$, the problem of checking robustness of a program $P$ is reduced to checking whether there exists a trace $tr \in Tr_{SI}(P) \setminus Tr_{SER}(P)$.

## 5 Reducing Robustness against $SI$ to SC Reachability

A trace which is not serializable must contain at least an issue and a commit event of the same transaction that don’t occur one after the other even after reordering of “independent” events. Thus, there must exist an event that occur between the two which is related to both events via the happens-before relation, forbidding the issue and commit to be adjacent. Otherwise, we can build another trace with the same happens-before where events are reordered such that the issue is immediately followed by the corresponding commit. The latter is a serializable trace which contradicts the initial assumption. We define a program instrumentation which mimics the delay of transactions by doing the writes on auxiliary variables which are not visible to other transactions. After the delay of a transaction, we track happens-before dependencies until we execute a transaction that does a “read” on one of the variables that the delayed transaction writes to (this would expose a read-write dependency to the commit event of the delayed transaction). While tracking happens-before dependencies we cannot execute a transaction that writes to a variable that the delayed transaction writes to since $SI$ forbids write-write conflicts between concurrent transactions.

Concretely, given a program $P$, we define an instrumentation of $P$ such that $P$ is not robust against $SI$ iff the instrumentation reaches an error state under serializability. The instrumentation uses auxiliary variables in order to simulate a single delayed transaction which we prove that it is enough for deciding robustness. Let $isu(p, t)$ be the issue event of the only delayed transaction. The process $p$ that delayed $t$ is called the Attacker. When the attacker finishes executing the delayed transaction it stops. Other processes that execute transactions afterwards are called Happens-Before Helpers.

The instrumentation uses two copies of the set of shared variables in the original program to simulate the delayed transaction. We use primed variables $x'$ to denote the second copy. Thus, when a process becomes the attacker, it will only write to the second copy that is not visible to other processes including the happens-before helpers. The writes made by the other processes including the happens-before helpers are made visible to all processes.
When the attacker delays the transaction $t$, it keeps track of the variables it accessed, in particular, it stores the name of one of the variables it writes to, $x$, it tracks every variable $y$ that it reads from and every variable $z$ that it writes to. When the attacker finishes executing $t$, and some other process wants to execute some other transaction, the underlying transaction must contain a write to a variable $y$ that the attacker reads from. Also, the underlying transaction must not write to a variable that $t$ writes to. We say that this process has joined happens-before helpers through the underlying transaction. While executing this transaction, we keep track of each variable that was accessed and the type of operation, whether it is a read or write. Afterward, in order for some other transaction to “join” the happens-before path, it must not write to a variable that $t$ writes to so it does not violate the fact that SI forbids write-write conflicts, and it has to satisfy one of the following conditions in order to ensure the continuity of the happens-before dependencies: (1) the transaction is issued by a process that has already another transaction in the happens-before dependency (program order dependency), (2) the transaction is reading from a shared variable that was updated by a previous transaction in the happens-before dependency (write-read dependency), (3) the transaction writes to a shared variable that was read by a previous transaction in the happens-before dependency (read-write dependency), or (4) the transaction writes to a shared variable that was updated by a previous transaction in the happens-before dependency (write-write dependency). We introduce a flag for each shared variable to mark the fact that the variable was read or written by a previous transaction.

Processes continue executing transactions as part of the chain of happens-before dependencies, until a transaction does a read on the variable $x$ that $t$ wrote to. In this case, we reached an error state which signals that we found a cycle in the transactional happens-before relation.

The instrumentation uses four varieties of flags: a) global flags (i.e., $HB$, $a_{tra}$, $a_{att}$), b) flags local to a process (i.e., $p.a$ and $p.hbh$), and c) flags per shared variable (i.e., $x.event$, $x.event'$, and $x.eventI$). We will explain the meaning of these flags along with the instrumentation. At the start of the execution, all flags are initialized to null ($\bot$).

Whether a process is an attacker or happens-before helper is not enforced syntactically by the instrumentation. It is set non-deterministically during the execution using some additional process-local flags. Each process chooses to set to true at most one of the flags $p.a$ and $p.hbh$, implying that the process becomes an attacker or happens-before helper, respectively. At most one process can be an attacker, i.e., set $p.a$ to true. In the following, we detail the instrumentation for read and write instructions of the attacker and happens-before helpers.

### 5.1 Instrumentation of the Attacker

Figure 3 lists the instrumentation of the write and read instructions of the attacker. Each process passes through an initial phase where it executes transactions that are visible immediately to all the other processes (i.e., they are not delayed), and then non-deterministically it can choose to delay a transaction at...
which point it sets the flag $a_{\text{ata}}$ to true. During the delayed transaction it chooses non-deterministically a write instruction to a variable $x$ and stores the name of this variable in the flag $a_{\text{ata}}$ (line (1)). The values written during the delayed transaction are stored in the primed variables and are visible only to the current transaction, in case the transaction reads its own writes. For example, given a variable $z$, all writes to $z$ from the original program are transformed into writes to the primed version $z'$ (line (3)). Each time, the attacker writes to $z$, it sets the flag $z.\text{event}' = 1$. This flag is used later by transactions from happens-before helpers to avoid writing to variables that the delayed transaction writes to.

A read on a variable, $y$, in the delayed transaction takes her value from the primed version, $y'$. In every read in the delayed transaction, we set the flag $y.\text{event}$ to id (line (1)) to be used latter in order for a process to join the happens-before helpers. Afterward, the attacker starts the happens-before path, and it sets the variable HB to true (line (2)) to mark the start of the happens. When the flag HB is set to true the attacker stops executing new transactions.

### 5.2 Instrumentation of the Happens-Before Helpers

The remaining processes, which are not the attacker, can become a happens-before helper. Figure 4 lists the instrumentation of write and read instructions of a happens-before helper. In a first phase, each process executes the original code until the flag $a_{\text{ata}}$ is set to true by the attacker. This flag signals the “creation” of the secondary copy of the shared-variables, which can be observed only by the attacker. At this point, the flag HB is set to true, and the happens-before helper process chooses non-deterministically a first transaction through which it wants to join the set of happens-before helpers, i.e., continue the happens-before dependency created by the existing happens-before helpers. When a process chooses a transaction, it makes a pledge (while executing the \textbf{begin} instruction) that during this transaction it will either read from a variable that was written to by another happens-before helper, write to a variable that was accessed (read or
Checking Robustness Against Snapshot Isolation

written) by another happens-before helper, or write to a variable that was read from in the delayed transaction. When the pledge is met, the process sets the flag \( p.hbh \) to true (lines (7) and (11)). The execution is blocked if a process does not keep its pledge (i.e., the flag \( p.hbh \) is null) at the end of the transaction. Note that the first process to join the happens-before helper must execute a transaction \( t \) which writes to a variable that was read from in the delayed transaction since this is the only way to build a happens-before between \( t \), and the delayed transaction (PO is not possible since \( t \) is not from the attacker, WR is not possible since \( t \) does not see the writes of the delayed transaction, and WW is not possible since \( t \) cannot write to a variable that the delayed transaction writes to). We use a flag \( x.event \) for each variable \( x \) to record the type (read \( ld \) or write \( st \)) of the last access made by a happens-before helper (lines (8) and (10)). During the execution of a transaction that is part of the happens-before dependency, we must ensure that the transaction does not write to variable \( y \) where \( y.event' \) is set to 1. Otherwise, the execution is blocked (line 9).

The happens-before helpers continue executing their instructions, until one of them reads from the shared variable \( x \) whose name was stored in \( a_{sta} \). This establishes a happens-before dependency between the delayed transaction and a “fictitious” store event corresponding to the delayed transaction that could be executed just after this read of \( x \). The execution doesn’t have to contain this store event explicitly since it is always enabled. Therefore, at the end of every transaction, the instrumentation checks whether the transaction read \( x \). If it is the case, then the execution stops and goes to an error state to indicate that this is a robustness violation. Notice that after the attacker stops, the only processes that are executing transactions are happens-before helpers, which is justified since when a process is not from a happens-before helper it implies that we cannot construct a happens-before dependency between a transaction of this process and the delayed transaction which means that the two transactions commute which in turn implies that this process’s transactions can be executed before executing the delayed transaction of the attacker.

\[
\begin{align*}
[l_1: r := x; \text{goto } l_2]_{\text{HbH}} &= \\
// \text{Read before the delayed transaction} \\
l_1: \text{assume HB} = \bot \land p.a = \bot; \text{goto } l_{11}; \\
l_{11}: r := x; \text{goto } l_2; \\
// \text{Read after the delayed transaction} \\
l_1: \text{assume HB} \neq \bot; \text{goto } l_{12}; \\
l_{12}: r := x; \text{goto } l_{13}; \\
l_{13}: \text{assume } x.event = \bot \land p.hbh = \bot; \text{goto } l_{14}; \\
l_{14}: p.hbh := \text{true}; \text{goto } l_2; \\
l_{13}: \text{assume } x.event = \bot; \text{goto } l_{15}; \\
l_{15}: x.event := \text{ld}; \text{goto } l_2; \\
l_{13}: \text{assume } x.event \neq \bot \lor p.hbh \neq \bot; \text{goto } l_2; \\
l_1: x := e; \text{goto } l_2; \\
// \text{Write before the delayed transaction} \\
l_1: \text{assume HB} = \bot \land a_{sta} = \bot; \text{goto } l_{11}; \\
l_{11}: x := e; \text{goto } l_2; \\
// \text{Write after the delayed transaction} \\
l_1: \text{assume HB} \neq \bot \land p.a = \bot; \text{goto } l_{12}; \\
l_{12}: \text{assume } x.event' \neq \bot; \text{assume false}; \\
l_{12}: \text{assume } x.event' = \bot; \text{goto } l_{13}; \\
l_{13}: x := e; \text{goto } l_{14}; \\
l_{14}: x.event := \text{st}; \text{goto } l_{15}; \\
l_{15}: \text{assume } x.event' \neq \bot \land p.hbh = \bot; \text{goto } l_{16}; \\
l_{16}: p.hbh := \text{true}; \text{goto } l_2; \\
l_{15}: \text{assume } x.event' = \bot \lor p.hbh \neq \bot; \text{goto } l_2;
\end{align*}
\]

Fig. 4: Instrumentation of Happens-Before Helpers.

\[
\begin{align*}
[l_1: r := x; \text{goto } l_2]_{\text{HbH}} &= \\
// \text{Write before the delayed transaction} \\
l_1: \text{assume HB} = \bot \land a_{sta} = \bot; \text{goto } l_{11}; \\
l_{11}: x := e; \text{goto } l_2; \\
// \text{Write after the delayed transaction} \\
l_1: \text{assume HB} \neq \bot \land p.a = \bot; \text{goto } l_{12}; \\
l_{12}: \text{assume } x.event' \neq \bot; \text{assume false}; \\
l_{12}: \text{assume } x.event' = \bot; \text{goto } l_{13}; \\
l_{13}: x := e; \text{goto } l_{14}; \\
l_{14}: x.event := \text{st}; \text{goto } l_{15}; \\
l_{15}: \text{assume } x.event' \neq \bot \land p.hbh = \bot; \text{goto } l_{16}; \\
l_{16}: p.hbh := \text{true}; \text{goto } l_2; \\
l_{15}: \text{assume } x.event' = \bot \lor p.hbh \neq \bot; \text{goto } l_2;
\end{align*}
\]
5.3 Correctness

The role of a process in an execution is chosen non-deterministically at runtime. Therefore, the final instrumentation of a given program $P$, denoted by $\lbrack P \rbrack$, is obtained by replacing each labeled instruction $\langle \text{linst} \rangle$ with the concatenation of the instrumentations corresponding to the attacker and the happens-before helpers, i.e., $\lbrack \langle \text{linst} \rangle \rbrack := \lbrack \langle \text{linst} \rangle \rbrack_{\text{A}} \lbrack \langle \text{linst} \rangle \rbrack_{\text{HB}}$

The following theorem states the correctness of the instrumentation.

**Theorem 2.** A program $P$ is not robust against SI iff $\lbrack P \rbrack$ reaches the error state.

If a program is not robust, this implies that its execution under SI results in a trace where the happens-before is cyclic. Which is possible only if the program contains at least one delayed transaction. In the proof, we show that is sufficient to search for executions that contain a single delayed transaction.

Notice that in the instrumentation of the attacker, the delayed transaction must contain a read and write instructions on different variables. Also, the transactions of the happens-before helpers must not contain a write to a variable that the delayed transaction writes to. The following corollary states the complexity of checking robustness for finite-state programs against snapshot isolation. It is a direct consequence of Theorem 2 and of previous results concerning the reachability problem in concurrent programs running over a sequentially-consistent memory, with a fixed [16] or parametric number of processes [21].

**Corollary 1.** Checking robustness of finite-state programs against snapshot isolation is PSPACE-complete when the number of processes is fixed and EXPSPACE-complete, otherwise.

It is important to note that the instrumentation can be easily extended to SQL (select/update) queries where a statement may include expressions over finite/infinite set of variables.

6 Proving Program Robustness

As a more pragmatic alternative to the reduction in the previous section, we define an approximated method for proving robustness which is inspired by Lipton’s reduction theory [17].

**Movers.** Given an execution $\tau = ev_1 \cdot \ldots \cdot ev_n$ of a program $P$ under serializability (where each event $ev_i$ corresponds to executing an entire transaction), we say that the event $ev_i$ moves right (resp., left) in $\tau$ if $ev_1 \cdot \ldots \cdot ev_{i-1} \cdot ev_{i+1} \cdot ev_i \cdot ev_{i+2} \cdot \ldots \cdot ev_n$ (resp., $ev_1 \cdot \ldots \cdot ev_{i-2} \cdot ev_i \cdot ev_{i-1} \cdot ev_{i+1} \cdot \ldots \cdot ev_n$) is also a valid execution of $P$, the process of $ev_i$ is different from the process of $ev_{i+1}$ (resp., $ev_{i-1}$), and both executions reach to the same end state $\sigma_n$. For an execution $\tau$, let $\text{instOf}_{\tau}(ev_i)$ denote the transaction that generated the event $ev_i$. A transaction $t$ of a program $P$ is a right (resp., left) mover if for all executions $\tau$ of $P$ under serializability, the event $ev_i$ with $\text{instOf}(ev_i) = t$ moves right (resp., left) in $\tau$.

---

1 That is, programs with a bounded number of variables taking values from a bounded data domain.
If a transaction \( t \) is not a right mover, then there must exist an execution \( \tau \) of \( P \) under serializability and an event \( ev_i \) of \( \tau \) with \( \text{instOf}(ev_i) = t \) that does not move right. This implies that there must exist another \( ev_{i+1} \) of \( \tau \) which caused \( ev_i \) to not be a right mover. Since \( ev_i \) and \( ev_{i+1} \) do not commute, then this must be because of either a write-read, write-write, or a read-write dependency. If \( t' = \text{instOf}(ev_{i+1}) \), we say that \( t \) is not a right mover because of \( t' \) and some dependency that is either write-read, write-write, or read-write. Notice that when \( t \) is not a right mover because of \( t' \) then \( t' \) is not a left mover because of \( t \).

We define \( M_{WR} \) as a binary relation between transactions such that \((t, t') \in M_{WR}\) when \( t \) is not a right mover because of \( t' \) and a write-read dependency. We define the relations \( M_{WW} \) and \( M_{RW} \) corresponding to write-write and read-write dependencies in a similar way.

**Read/Write-free transactions.** Given a transaction \( t \), we define \( t \setminus \{r\} \) as a variation of \( t \) where all the reads from shared variables are replaced with non-deterministic reads, i.e., \( \langle \text{reg} \rangle := \langle \text{var} \rangle \) statements are replaced with \( \langle \text{reg} \rangle := \star \) where \( \star \) denotes non-deterministic choice. We also define \( t \setminus \{w\} \) as a variation of \( t \) where all the writes to shared variables in \( t \) are disabled. Intuitively, recalling the reduction to SC reachability in Section 5, \( t \setminus \{r\} \) simulates the delay of a transaction by the Attacker, i.e., the writes are not made visible to other processes, and \( t \setminus \{r\} \) approximates the commit of the delayed transaction which only applies a set of writes.

**Commutativity dependency graph.** Given a program \( P \), we define the commutativity dependency graph as a graph where vertices represent transactions and their read/write-free variations. Two vertices which correspond to the original transactions in \( P \) are related by a program order edge, if they belong to the same process. The other edges in this graph represent the “non-mover” relations \( M_{WR}, M_{WW}, \) and \( M_{RW} \).

Given a program \( P \), we say that the commutativity dependency graph of \( P \) contains a non-mover cycle if there exist a set of transactions \( t_0, t_1, \ldots, t_n \) of \( P \) such that the following hold:

(a) \((t_0', t_1) \in M_{RW}\) where \( t_0' \) is the write-free variation of \( t_0 \) and \( t_1 \) does not write to a variable that \( t_0 \) writes to;

(b) for all \( i \in [1, n] \), \((t_i, t_{i+1}) \in (PO \cup M_{WR} \cup M_{WW} \cup M_{RW})\), \( t_i \) and \( t_{i+1} \) do not write to a shared variable that \( t_0 \) writes to;

(c) \((t_n, t_0') \in M_{RW}\) where \( t_0' \) is the read-free variation of \( t_0 \) and \( t_n \) does not write to a variable that \( t_0 \) writes to.

A non-mover cycle approximates an execution of the instrumentation defined in Section 5 in between the moment that the Attacker delays a transaction \( t_0 \) (which here corresponds to the write-free variation \( t_0' \)) and the moment where \( t_0 \) gets committed (the read-free variation \( t_0' \)).

The following theorem shows that the acyclicity of the commutativity dependency graph of a program implies the robustness of the program. Actually, the notion of robustness in this theorem relies on a slightly different notion of trace where store-order and write-order dependencies take into account values, i.e., store-order relates only writes writing different values and the write-order
relates a read to the oldest write (w.r.t. execution order) writing its value. This relaxation helps in avoiding some harmless robustness violations due to for instance, two transactions writing the same value to some variable.

**Theorem 3.** For a program $P$, if the commutativity dependency graph of $P$ does not contain non-mover cycles, then $P$ is robust.

### 7 Experiments

To test the applicability of our robustness checking algorithms, we have considered a benchmark of 10 applications extracted from the literature related to weakly consistent databases in general. A first set of applications are open source projects that were implemented to be run over the Cassandra database, extracted from [10]. The second set of applications is composed of: TPC-C [23], an on-line transaction processing benchmark widely used in the database community, SmallBank, a simplified representation of a banking application [2], FusionTicket, a movie ticketing application [15], Auction, an online auction application [3], and Courseware, a course registration service extracted from [13, 18].

A first experiment concerns the reduction of robustness checking to SC reachability. For each application, we have constructed a client (i.e., a program composed of transactions defined within that application) with a fixed number of processes (at most 3) and a fixed number of transactions (between 3 and 7 transactions per process). We have encoded the instrumentation of this client, defined in Section 5 in the Boogie programming language [3] and used the Civl verifier [14] in order to check whether the assertions introduced by the instrumentation are violated (which would represent a robustness violation). Note that since clients are of fixed size, this requires no additional assertions/invariants (it is an instance of bounded model checking). The results are reported in Table 1. We have found two of the applications, Courseware and SmallBank, to not be robust against snapshot isolation. The violation in Courseware is caused by transactions RemoveCourse and EnrollStudent that execute concurrently, RemoveCourse removing a course that has no registered student and EnrollStudent registering a new student to the same course. We get an invalid state where a student is registered for a course that was removed. SmallBank’s violation contains transactions Balance, TransactSaving, and WriteCheck. One process executes WriteCheck where it withdraws an amount from the checking account after checking that the sum of the checking and savings accounts is bigger than this amount. Concurrently, a second process executes TransactSaving where it withdraws an amount from the saving account after checking that it is smaller than the amount in the savings account. Afterwards, the second process checks the contents of both the checking and saving accounts. We get an invalid state where the sum of the checking and savings accounts is negative.

Since in the first experiment we consider fixed clients, the lack of assertion violations doesn’t imply that the application is robust (this instantiation of our reduction can only be used to reveal robustness violations). Thus, a second experiment concerns the robustness proof method based on commutativity
Table 1: An overview of the analysis results. CDG stands for commutativity dependency graph. The columns PO and PT show the number of proof obligations and proof time in second, respectively. T stands for trivial when the application has only read-only transactions.

| Application     | #Transactions | Robustness | Reachability Analysis | CDG Analysis |
|-----------------|---------------|------------|-----------------------|--------------|
|                 |               | PO         | PT                    | PO           | PT           |
| Auction         | 4             | ✓          | 70                    | 0.3          | 20           | 0.5         |
| Courseware      | 5             | X          | 59                    | 0.37         | na           | na          |
| FusionTicket    | 4             | ✓          | 72                    | 0.3          | 34           | 0.5         |
| SmallBank       | 5             | X          | 48                    | 0.28         | na           | na          |
| TPC-C           | 5             | ✓          | 54                    | 0.7          | 82           | 3.7         |
| Cassioq-Core    | 8             | ✓          | 173                   | 0.55         | 104          | 2.9         |
| Currency-Exchange| 6            | ✓          | 88                    | 0.35         | 26           | 3.5         |
| PlayList        | 14            | ✓          | 99                    | 4.63         | 236          | 7.3         |
| RoomStore       | 5             | ✓          | 85                    | 0.3          | 22           | 0.5         |
| Shopping-Cart   | 4             | ✓          | 58                    | 0.25         | T            | T           |

dependency graphs (Section 6). For the applications that were not identified as non-robust by the previous method, we have used Civl to construct their commutativity dependency graphs, i.e., identify the “non-mover” relations $M_{WR}$, $M_{WW}$, and $M_{RW}$ (Civl allows to check whether some code fragment is a left/right mover). In all cases, the graph didn’t contain non-mover cycles, which allows to conclude that the applications are robust.

The experiments show that our results can be used for finding violations and proving robustness, and that they apply to a large set of interesting examples. Note that the reduction to SC and the proof method based on commutativity dependency graphs are valid for programs with SQL (select/update) queries.

8 Related Work

Decidability and complexity of robustness has been investigated in the context of relaxed memory models such as TSO and Power [6, 8, 12]. The work we present in this paper borrows some high-level principles from [6] which addresses the robustness against TSO. We reuse only the high-level methodology of characterizing minimal violations according to some measure and defining reductions to SC reachability using a program instrumentation. Instantiating this methodology in our context is however very different. Several fundamental differences between our work and [6] are:

- SI and TSO admit different sets of relaxations and SI is a model of transactional databases.
- We use a different notion of measure: the measure in [6] counts the number of events between a write issue and a write commit while our notion of measure counts the number of delayed transactions. This is a first reason for which the proof techniques in [6] don’t extend to our context.
- Transactions induce more complex traces: two transactions might be related by several dependency relations since each transaction may contain multiple
reads and writes to different locations. In the case of TSO, each action is
either a read or a write to a specific location, and two events are related by a
single dependency relation. Moreover, the number of dependencies between
two transactions depends on the execution since the set of reads/writes in a
transaction evolves dynamically.

Other works, [6, 12], define decision procedures for robustness which are based
on the theory of regular languages and do not extend to infinite-state programs
like in our case.

As far as we know, the decidability and complexity issues of the robust-
ness checking problem have never been investigated in the context of transac-
tional programs. Our work is the first one that establishes results on this topic,
considering the important model of snapshot isolation. The existing work on the
verification of robustness for transactional programs provide either over- or under-
approximate analyses for checking it. Our commutativity dependency graphs are sim-
ilar to the static dependency graphs used
in [8, 9, 10, 11], but they are more precise, i.e., reducing the number of false
alarms. The static dependency graphs record happens-before dependencies be-
tween transactions based on a syntactic approximation of the variables read-
/written by a transaction. For example, our techniques are able to prove that the
program in Figure 5 is robust, while this is not possible using static dependency
graphs. The latter would contain a dependency from transaction \( t_1 \) to \( t_2 \) and one
from \( t_2 \) to \( t_1 \) just because syntactically, each of the two transactions reads both
variables and may write to one of them. Our commutativity dependency graphs
take into account the semantics of those two transactions and don’t include this
happens-before cycle. Other over- and under-approximate analyses have been
proposed in [19]. They are based on encoding program executions into first or-
der logic, bounded-model checking for the under-approximate analysis, and a
sound check for proving a cut-off bound on the size of the happens-before cycles
possible in the executions of a program, for the over-approximate analysis. The
latter is strictly less precise than our method based on commutativity depen-
dency graphs. For instance, extending the TPC-C application with additional
transactions will make the method in [10] fail while our method will succeed in
proving robustness (the three transactions are for adding a new product, adding
a new warehouse based on the number of customers and warehouses, and adding
a new customer, respectively).

Finally, the idea of using Lipton’s reduction theory for checking robustness
has been also used in the context of the TSO memory model [7], but the tech-
niques are completely different, e.g., the TSO technique considers each update
in isolation and doesn’t consider non-mover cycles like in our commutativity
dependency graphs.
Bibliography

[1] Adya, A.: Weak consistency: A generalized theory and optimistic implementations for distributed transactions. Ph.D. thesis (1999)

[2] Alomari, M., Cahill, M.J., Fekete, A., Röhm, U.: The cost of serializability on platforms that use snapshot isolation. In: Alonso, G., Blakeley, J.A., Chen, A.L.P. (eds.) Proceedings of the 24th International Conference on Data Engineering, ICDE 2008, April 7-12, 2008, Cancún, Mexico. pp. 576–585. IEEE Computer Society (2008)

[3] Barnett, M., Chang, B.E., DeLine, R., Jacobs, B., Leino, K.R.M.: Boogie: A modular reusable verifier for object-oriented programs. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.P. (eds.) Formal Methods for Components and Objects, 4th International Symposium, FMCO 2005, Amsterdam, The Netherlands, November 1-4, 2005, Revised Lectures. Lecture Notes in Computer Science, vol. 4111, pp. 364–387. Springer (2005)

[4] Berenson, H., Bernstein, P.A., Gray, J., Melton, J., O’Neil, E.J., O’Neil, P.E.: A critique of ANSI SQL isolation levels. In: Carey, M.J., Schneider, D.A. (eds.) Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data, San Jose, California, USA, May 22-25, 1995. pp. 1–10. ACM Press (1995)

[5] Bernardi, G., Gotsman, A.: Robustness against consistency models with atomic visibility. In: Desharnais, J., Jagadeesan, R. (eds.) 27th International Conference on Concurrency Theory, CONCUR 2016, August 23-26, 2016, Québec City, Canada. LIPIcs, vol. 59, pp. 7:1–7:15. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2016)

[6] Bouajjani, A., Derevenetc, E., Meyer, R.: Checking and enforcing robustness against TSO. In: Felleisen, M., Gardner, P. (eds.) Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings. Lecture Notes in Computer Science, vol. 7792, pp. 533–553. Springer (2013)

[7] Bouajjani, A., Enea, C., Mutluergil, S.O., Tasiran, S.: Reasoning about TSO programs using reduction and abstraction. In: Chockler, H., Weissenbacher, G. (eds.) Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part II. Lecture Notes in Computer Science, vol. 10982, pp. 336–353. Springer (2018)

[8] Bouajjani, A., Meyer, R., Möhlmann, E.: Deciding robustness against total store ordering. In: Aceto, L., Henzinger, M., Sgall, J. (eds.) Automata, Languages and Programming - 38th International Colloquium, ICALP 2011, Zurich, Switzerland, July 4-8, 2011, Proceedings, Part II. Lecture Notes in Computer Science, vol. 6756, pp. 428–440. Springer (2011)

[9] Brutschy, L., Dimitrov, D., Müller, P., Vechev, M.T.: Serializability for eventual consistency: criterion, analysis, and applications. In: Castagna, G., Gor-
Brutschy, L., Dimitrov, D., Müller, P., Vechev, M.T.: Static serializability analysis for causal consistency. In: Foster, J.S., Grossman, D. (eds.) Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018. pp. 90–104. ACM (2018)

Cerone, A., Gotsman, A.: Analysing snapshot isolation. J. ACM 65(2), 11:1–11:41 (2018)

Derevenetc, E., Meyer, R.: Robustness against power is pspace-complete. In: Esparza, J., Fraigniaud, P., Husfeldt, T., Koutsoupias, E. (eds.) Automata, Languages, and Programming - 41st International Colloquium, ICALP 2014, Copenhagen, Denmark, July 8-11, 2014, Proceedings, Part II. Lecture Notes in Computer Science, vol. 8573, pp. 158–170. Springer (2014)

Gotsman, A., Yang, H., Ferreira, C., Najafzadeh, M., Shapiro, M.: ’cause i’m strong enough: reasoning about consistency choices in distributed systems. In: Bodík, R., Majumdar, R. (eds.) Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016. pp. 371–384. ACM (2016)

Hawblitzel, C., Petrank, E., Qadeer, S., Tasiran, S.: Automated and modular refinement reasoning for concurrent programs. In: Kroening, D., Pasareanu, C.S. (eds.) Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part II. Lecture Notes in Computer Science, vol. 9207, pp. 449–465. Springer (2015)

Holz, B., Bornholt, J., Zhang, I., Ports, D.R.K., Oskin, M., Ceze, L.: Disciplined inconsistency with consistency types. In: Aguilera, M.K., Cooper, B., Dao, Y. (eds.) Proceedings of the Seventh ACM Symposium on Cloud Computing, Santa Clara, CA, USA, October 5-7, 2016. pp. 279–293. ACM (2016)

Kozen, D.: Lower bounds for natural proof systems. In: 18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977. pp. 254–266. IEEE Computer Society (1977)

Lipton, R.J.: Reduction: A method of proving properties of parallel programs. Commun. ACM 18(12), 717–721 (1975)

Nagar, K., Jagannathan, S.: Automated detection of serializability violations under weak consistency. In: Schewe, S., Zhang, L. (eds.) 29th International Conference on Concurrency Theory, CONCUR 2018, September 4-7, 2018, Beijing, China. LIPIcs, vol. 118, pp. 41:1–41:18. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2018)
[19] Nagar, K., Jagannathan, S.: Automatic detection of serializability violations under weak consistency. In: 29th Intern. Conf. on Concurrency Theory (CONCUR’18) (September 2018), to appear
[20] Papadimitriou, C.H.: The serializability of concurrent database updates. J. ACM 26(4), 631–653 (1979)
[21] Rackoff, C.: The covering and boundedness problems for vector addition systems. Theor. Comput. Sci. 6, 223–231 (1978)
[22] Shasha, D.E., Snir, M.: Efficient and correct execution of parallel programs that share memory. ACM Trans. Program. Lang. Syst. 10(2), 282–312 (1988)
[23] TPC: Tech. rep., Transaction Processing Performance Council (February 2010), http://www.tpc.org/tpc_documents_current_versions/pdf/tpc-c_v5.11.0.pdf
A Programs

A.1 Program Syntax

We consider a simple programming language grammar which is defined in Figure 6. A program is parallel composition of processes distinguished using a set of identifiers $P$. Each process is a sequence of transactions and each transaction is a sequence of labeled instructions. Each transaction starts with a begin instruction and finishes with a commit instruction. Each other instruction is either an assignment to a process-local register from a set $R$ or to a shared variable from a set $V$, or an assume statement. The read/write assignments use values from a data domain $D$. An assignment to a register $\langle \text{reg} \rangle := \langle \text{var} \rangle$ is called a read of $\langle \text{var} \rangle$ and an assignment to a shared variable $\langle \text{var} \rangle := \langle \text{reg-expr} \rangle$ is called a write to $\langle \text{var} \rangle$ ($\langle \text{reg-expr} \rangle$ is an expression over registers whose syntax we leave unspecified since it is irrelevant for our development). The assume $\langle \text{bexpr} \rangle$ blocks the process if the Boolean expression $\langle \text{bexpr} \rangle$ over registers is false.

$$\langle \text{prog} \rangle ::= \text{program} \langle \text{process} \rangle^*$$
$$\langle \text{process} \rangle ::= \text{process} \langle \text{pid} \rangle \text{regs} \langle \text{reg} \rangle^*$$
$$\langle \text{ltxn} \rangle ::= \langle \text{binst} \rangle \langle \text{linst} \rangle^* \langle \text{einst} \rangle$$
$$\langle \text{binst} \rangle ::= \langle \text{label} \rangle : \text{begin}; \text{goto} \langle \text{label} \rangle;$$
$$\langle \text{einst} \rangle ::= \langle \text{label} \rangle : \text{end}; \text{goto} \langle \text{label} \rangle;$$
$$\langle \text{linst} \rangle ::= \langle \text{label} \rangle : \langle \text{inst} \rangle ; \text{goto} \langle \text{label} \rangle;$$
$$\langle \text{inst} \rangle ::= \langle \text{reg} \rangle := \langle \text{var} \rangle$$
$$\quad | \langle \text{var} \rangle := \langle \text{reg-expr} \rangle$$
$$\quad | \text{assume} \langle \text{bexpr} \rangle$$

Fig. 6: Program syntax. $a^*$ indicates zero or more occurrences of $a$. $\langle \text{pid} \rangle$, $\langle \text{reg} \rangle$, $\langle \text{label} \rangle$, and $\langle \text{var} \rangle$ represent a process identifier, a register, a label, and a shared variable, respectively. $\langle \text{reg-expr} \rangle$ is an expression over registers while $\langle \text{bexpr} \rangle$ is a Boolean expression over registers.

A.2 Program Semantics Under SnapShot Isolation

The semantics of a program under SI is defined as follows. The shared variables are stored in a central memory and each process keeps a replicated copy of the central memory. A process starts a transaction by discarding its local copy and fetching the values of the shared variables from the central memory. When a process commits a transaction, it merges its local copy of the shared variables
with the one stored in the central memory in order to make its updates visible to all processes. During the execution of a transaction, the process stores the writes to shared variables only in its local copy and reads only from its local copy. When a process merges its local copy with the centralized one, it is required that there were no concurrent updates that occurred after the last fetch from the central memory to a shared variable that was updated by the current transaction. Otherwise, the transaction is aborted and its effects discarded.

Thus, a program configuration is a tuple \( gs = (ls, tstamp, Log) \) where \( ls : \mathbb{P} \to \mathbb{S} \) associates a local state in \( \mathbb{S} \) to each process in \( \mathbb{P} \), \( tstamp : \mathbb{V} \to \mathbb{T} \) stores the largest timestamp for each shared variable, and \( Log : \mathbb{V} \to \mathbb{D} \) holds the global valuation of shared variables. A local state is a tuple \( (pc, store, log, rval) \) where \( pc \in \text{Lab} \) is the program counter, i.e., the label of the next instruction to be executed, \( store : \mathbb{V} \to \mathbb{D} \) is the local valuation of the shared variables, \( log : \mathbb{V} \to \{⊥, 1\} \) is a local log which marks shared variables which were updated in a transaction, and \( rval : \mathbb{R} \to \mathbb{D} \) is the valuation of the local registers. For a local state \( s \), we use \( s.pc \) to denote the program counter component of \( s \), and similarly for all the other components of \( s \). Given a transaction \( t \in \mathbb{T} \times \mathbb{T} \), we use \( t.st \) to denote the start time of transaction \( t \) and \( t.ct \) to denote the commit time of \( t \). Before merging \( store \) with \( Log \), after executing a transaction \( t \), we check that for every variable \( x \) that \( t \) writes to \( (log(x) \neq ⊥) \) we have that \( tstamp(x) < t.st \) (i.e., there were no concurrent write to \( x \)). Then, we store the value of \( store(x) \) for every variable \( x \) that \( t \) writes to \( (log(x) \neq ⊥) \) in \( Log(x) \). Also, for every variable \( x \) that \( t \) writes to, we store \( t.ct \) in \( tstamp(x) \).

Then, the semantics of a program \( \mathcal{P} \) under snapshot isolation consistency model is defined using a labeled transition system (LTS) \( [\mathcal{P}]_{\text{SI}} = (\mathbb{C}, \mathbb{E}v, gs_0, \to) \) where \( \mathbb{C} \) is the set of program configurations, \( \mathbb{E}v \) is a set of transition labels called \( \text{events} \), \( gs_0 \) is the initial program configuration, and \( \to \subseteq \mathbb{C} \times \mathbb{E}v \times \mathbb{C} \) is the transition relation. The set of events under \( \text{SI} \) is defined as follow.
\[
\mathbb{E}v = \{ \text{begin}(p, t), \text{ld}(p, t, x, v), \text{isu}(p, t, x, v), \text{com}(p, t) : p \in \mathbb{P}, t \in \mathbb{T} \times \mathbb{T}, x \in \mathbb{V}, v \in \mathbb{D} \}
\]
where \( \text{begin} \) and \( \text{com} \) label transitions corresponding to the start and the commit of a transaction, respectively. \( \text{isu} \) and \( \text{ld} \) label transitions corresponding to writing, resp., reading, a shared variable during some transaction.

The transition relation \( \to \) is defined in Figure 7. For readability, the events labeling a transition are written on top of \( \to \). A \( \text{begin} \) transition resets the local valuation of the shared variables and fetches their values from the central memory. A \( \text{com} \) transition applies the writes performed in a transaction to the central memory by merging the contents of the local copy \( store \) with the central memory \( Log \). An \( \text{ld} \) transition reads the value of a shared-variable from the local copy \( store \) while an \( \text{isu} \) transition applies a new write to the local copy \( store \).

An execution of program \( \mathcal{P} \), under snapshot isolation, is a sequence of events \( ev_1 \cdot ev_2 \cdot \ldots \) labeling the transitions, such that there exists a sequence of configurations \( gs_0 \cdot gs_1 \cdot \ldots \) where \( gs_0 \) is the initial configuration before \( \mathcal{P} \) starts execution and \( gs_{i-1} \xrightarrow{ev_i} gs_i \) is a valid transition for \( i > 1 \) of transitions. The set of executions of \( \mathcal{P} \) under \( \text{SI} \) is denoted by \( \mathbb{E}x_{\text{SI}}(\mathcal{P}) \).
begin \in \text{inst}(ls(p), pc) \quad \text{img}(ls, \text{tstamp}) < \text{t.st} \quad s = ls(p)[\text{log} \rightarrow r, \text{store} \rightarrow \text{Log}, pc \rightarrow \text{next}(pc)] \\
\quad (ls, \text{tstamp}, \text{Log}, \text{lk}) \xrightarrow{\text{begin}(p, t, v)} (ls[p \rightarrow s], \text{tstamp}, \text{Log}, \text{lk}) \\
r := x \in \text{inst}(ls(p), pc) \quad ls(p).\text{store}[x] = v \quad rval = ls(p).\text{rval}[r \rightarrow v] \\
s = ls(p)[rval \rightarrow rval, pc \rightarrow \text{next}(pc)] \\
\quad (ls, \text{tstamp}, \text{Log}, \text{lk}) \xrightarrow{\text{if}(p, t, x, v)} (ls[p \rightarrow s], \text{tstamp}, \text{Log}, \text{lk}) \\
x := v \in \text{inst}(ls(p), pc) \quad log = ls(p),\text{log}[x \rightarrow 1] \quad \text{store} = ls(p).\text{store}[x \rightarrow v] \\
s = ls(p)[\text{log} \rightarrow log, \text{store} \rightarrow \text{store}, pc \rightarrow \text{next}(pc)] \\
\quad (ls, \text{tstamp}, \text{Log}, \text{lk}) \xrightarrow{\text{inst}(p, t, x, v)} (ls[p \rightarrow s], \text{tstamp}, \text{Log}, \text{lk}) \\
end \in \text{inst}(ls(p), pc) \quad \forall x \in V. log[x] = \bot \lor \text{tstamp}(x) < \text{t.ct} \\
\quad Log = \text{Log}(x \rightarrow \text{store}[x] : x \in V, log[x] \neq \bot) \\
\quad \text{tstamp} = \text{tstamp}(x \rightarrow \text{ct.st} : x \in V, log[x] \neq \bot) \\
\quad s = ls(p)[pc \rightarrow \text{next}(pc)] \\
\quad (ls, \text{tstamp}, \text{Log}, \text{lk}) \xrightarrow{\text{com}(p, t, x, v)} (ls[p \rightarrow s], \text{Log} \rightarrow \text{Log}, \text{tstamp} \rightarrow \text{tstamp}, \text{lk})

Fig. 7: The set of transition rules defining snapshot isolation semantics model.
We assume that all the events which come from the same transaction use a unique transaction identifier \( t : (st, ct) \) that has two components. For a function \( f \), we use \( f[a \rightarrow b] \) to denote a function \( g \) such that \( g(c) = f(c) \) for all \( c \neq a \) and \( g(a) = b \). The function \text{inst} returns the set of instructions labeled by some given label while \text{next} gives the next instruction to execute.

B Proofs of Section 5: Characterizations of SI Trace-Robustness

In this section, we describe the proof of Theorem 2. We first reduce the robustness of a program \( P \) against SI to the existence of some execution trace \( tr \in \mathcal{T}_{\text{SI}}(P) \) \( \setminus \mathcal{T}_{\text{SER}}(P) \) that has a specific shape. We call a trace \( tr \in \mathcal{T}_{\text{SI}}(P) \) \( \setminus \mathcal{T}_{\text{SER}}(P) \), an anomaly. Then, we show that the anomaly of a particular shape is equivalent to an execution of the instrumented program reaching an error state.

First, we give an auxiliary lemma about the happens-before relation (between events). In the remaining of this section, we use \( \mathcal{HB}^1 \) to denote the happens-before without the transitive closure, i.e., \( \mathcal{HB}^1 = (\mathcal{PO} \cup \mathcal{WW} \cup \mathcal{WR} \cup \mathcal{RW} \cup \mathcal{STO}) \).

To decide if two events in a trace are “independent” (or commutative) we use the information about the existence of a happens-before relation between the events. If two events are not related by happens-before then they can be swapped while preserving the same happens-before. Thus, we extend the happens-before relation to obtain the happens-before through relation as follows:

Definition 2 (\[8\]). Let \( \tau = \alpha \cdot a \cdot \beta \cdot b \cdot \gamma \) be a trace where \( a \) and \( b \) are events (or atomic macro events), and \( \alpha, \beta, \) and \( \gamma \) are sequences of events (or atomic macro events) under a semantics SI. We say that \( a \) happens-before \( b \) through \( \beta \) if there is a non empty sub-sequence \( c_1 \cdots c_n \) of \( \beta \) that satisfies:

\[ c_i \rightarrow_{\mathcal{HB}^1} c_{i+1} \quad \text{for all } i \in [0, n] \]
where $c_0 = a$, $c_{n+1} = b$.

We deduce from the definition of serializable traces, that an anomaly trace must contain at least an issue and a commit events of the same transaction that are related via the happens-before through relation. Otherwise, we can build another trace with the same happens-before where events are reordered such that every issue $isu(p, t)$ is immediately followed by the corresponding commit $com(p, t)$. The latter is a serializable trace which contradicts the initial assumption.

**Lemma 1.** Given an anomaly $τ$, there must exist a transaction $t$ such that $τ = α · isu(p, t) · β · com(p, t) · γ$ and $isu(p, t)$ happens before $com(p, t)$ through $β$.

Given an anomaly of the from $τ = α · isu(p, t) · β · com(p, t) · γ$, we call $t$ a delayed transaction in the trace $τ$ when $isu(p, t)$ happens before $com(p, t)$ through $β$.

For an anomaly $τ$, the number of delays, denoted by $#(τ)$, in $τ$ is the total number of delayed transactions in the trace.

$$ #(τ) = # t is a delayed transaction in τ$$

**Definition 3 (Minimal anomaly).** An anomaly $τ$ is called minimal if it has the least number of delays among all possible anomalies (for a given program $P$).

Given an anomaly $τ = α · isu(p, t) · β · com(p, t) · γ$, and assuming that $t$ is the first delayed transaction in $τ$ (w.r.t. the order between issue events of delayed transactions) and that $τ$ is a minimal anomaly, the following lemma shows that we can assume w.l.o.g. that $γ$ is empty.

**Lemma 2.** Let $τ = α · isu(p, t) · β · com(p, t) · γ$ be a minimal anomaly such that $isu(p, t)$ happens-before $com(p, t)$ through $β$. Then, $τ' = α · isu(p, t) · β · com(p, t)$ is also a minimal anomaly.

**Proof.** We can notice that after executing the event $com(p, t)$, we obtain a cycle in the $HB_t$ relation. Thus, $τ'$ is already an anomaly not serializable.

The following result relates SI robustness problem to finding certain anomaly trace which is a minimal anomaly where only a single transaction is delayed.

**Theorem 4.** A program $P$ is not robust under $SI$ iff there exists an anomaly $τ$ under $SI$ such that the following must hold:

$$ τ = α · isu(p, t) · β · com(p, t)$$

(a) $isu(p, t)$ is the issue of the only delayed transaction in $τ$; (Lemmas 4 and 3);
(b) $isu(p, t)$ happens before $com(p, t)$ through $β$ (Lemma 5);
(c) for any event $a ∈ β$, we have that $(isu(p, t), a) ∈ HB$ and $(a, com(p, t)) ∈ HB$ (Lemma 5);
(d) there exist events $a$ and $b$ in $β$ such that $(isu(p, t), a) ∈ RW(x)$ and $(b, com(p, t)) ∈ RW(y)$ with $x ≠ y$ (Lemma 5);
(e) all delayed transactions in $β$ don’t write to shared variables that $t$ writes to (Lemma 4).
Note that in certain cases the events \(a\) and \(b\) can be identical and \(\beta = a\).

Figure 8 shows two examples of anomalies of the form given in Theorem 4.

In the following, we give the lemmas that constitute Theorem 4.

An important property in SI semantics is that of conflict-free, and since the event \(\text{com}(p, t)\) is successfully executed only if there are no concurrent writes that were committed after \(\text{isu}(p, t)\). Thus, for every event \(\text{com}(p_0, t_0)\) in \(\beta\), \(\text{com}(p_0, t_0)\) don’t write to a shared variable that \(\text{com}(p, t)\) writes to.

**Lemma 3.** Let \(\tau = \alpha \cdot \text{isu}(p, t) \cdot \beta \cdot \text{com}(p, t)\) be a minimal anomaly such that \(\text{isu}(p, t)\) happens-before \(\text{com}(p, t)\) through \(\beta\). Then, for every \(a \in \beta\), \(a\) does not write to a shared variable that \(\text{com}(p, t)\) writes to.

The following lemma shows that we must have an event \(\text{isu}(p', t')\) in \(\beta\) such that \(\text{isu}(p', t')\) happens before \(\text{com}(p, t)\).

**Lemma 4.** Let \(\tau = \alpha \cdot \text{isu}(p, t) \cdot \beta \cdot \text{com}(p, t)\) be a minimal anomaly such that \(\text{isu}(p, t)\) happens-before \(\text{com}(p, t)\) through \(\beta\). Then, there must exist an event \(\text{isu}(p', t')\) in \(\beta\) such that \(\text{isu}(p', t')\) happens before \(\text{com}(p, t)\), i.e., \((\text{isu}(p', t'), \text{com}(p, t))\) \(\in\) \(\text{HB}\).

**Proof.** Suppose by contradiction that \(\beta\) does not contain an event \(\text{isu}(p', t')\) such that \(\text{isu}(p', t')\) happens before \(\text{com}(p, t)\). Then, \(\text{com}(p, t)\) can be swapped with every \(\text{isu}(p', t')\) event in \(\beta\). Thus, we obtain that the two events \(\text{isu}(p, t)\) and \(\text{com}(p, t)\) are adjacent which means that \(t\) is no longer a delayed transaction which is a contradiction. Therefore, \(\beta\) does contain an event \(\text{isu}(p', t')\) such that \((\text{isu}(p', t'), \text{com}(p, t))\) \(\in\) \(\text{HB}\).

Next lemma shows that we can always obtain a minimal anomaly trace \(\tau = \alpha \cdot \text{isu}(p, t) \cdot \beta \cdot \text{com}(p, t)\) where \(\beta\) contains no delayed transaction. We show that if it were to have a delayed transaction \(t_0\) in \(\beta\), then it is possible to obtain a new anomaly where either \(t\) is not delayed or \(t_0\) is not delayed, and obtain a new anomaly with a smaller number of delayed transactions which contradicts the minimality assumption.

**Lemma 5.** Let \(\tau = \alpha \cdot \text{isu}(p, t) \cdot \beta \cdot \text{com}(p, t)\) be a minimal anomaly. Then, \(\beta\) does not contain delayed transactions.
Proof. We suppose by contradiction that \( \beta \) contains a delayed transaction \( t_0 \) issued by a process \( p_0 \).

It is important to notice that there must exist \( \beta' \subset \beta \) and \( \text{com}(p_0, t_0) \in \beta \) such that \( \text{isu}(p_0, t_0) \) happens before \( \text{com}(p_0, t_0) \) through \( \beta \). Otherwise, we can commute the events until \( \text{com}(p_0, t_0) \) occurs just after \( \text{isu}(p_0, t_0) \) and in this case transaction \( t_0 \) is not delayed by \( p_0 \). Thus, \( \tau \) is of the form \( \tau = \alpha \cdot \text{isu}(p, t) \cdot \beta_1 \cdot \text{isu}(p_0, t_0) \cdot \beta_2 \cdot \text{com}(p, t) \).

Notice that we can build happens-before cycle from \( \text{isu}(p_0, t_0) \) to \( \text{com}(p_0, t_0) \). Also, since in \( \beta_1 \cdot \text{isu}(p_0, t_0) \cdot \beta_2 \cdot \text{com}(p, t) \) no transaction depends on \( \text{isu}(p, t) \). Thus, we can safely remove \( \text{isu}(p, t) \) and its associated commit event \( \text{com}(p, t) \) and obtains: \( \tau' = \alpha \cdot \beta_1 \cdot \text{isu}(p_0, t_0) \cdot \beta_2 \cdot \text{com}(p_0, t_0) \cdot \beta_2 \). which is an anomaly because of the cycle formed as \( \text{isu}(p_0, t_0) \) happens-before \( \text{com}(p_0, t_0) \) through \( \beta' \).

In \( \tau' \), transaction \( t \) was not delayed, therefore, \( \tau' \) has less number of delays than \( \tau \). Thus, \( \tau \) is not a minimal anomaly, a contradiction to our hypothesis.

The next lemma characterizes the relation between the first delayed transaction and the commit of the underlying transaction. It shows the type of the first and last happens-before relations in the happens-before path between the issue of the only delayed transaction and its corresponding commit.

**Lemma 6.** Let \( \tau = \alpha \cdot \text{isu}(p, t) \cdot \beta \cdot \text{com}(p, t) \) be a minimal anomaly under SI. Then, the following must hold: There exist \( (p_0, t_0), (p_1, t_1) \in \beta \) where \( (\text{isu}(p, t), (p_0, t_0)) \in \text{RW} \), and \( ((p_1, t_1), \text{com}(p, t)) \in \text{RW} \).

Proof. We have that since \( \text{isu}(p, t) \) happens-before \( \text{com}(p, t) \) through \( \beta \) and \( \beta \) does not contain a delayed transaction. Then, there must exist \( (p_0, t_0) \) and \( (p_1, t_1) \) in \( \beta \) such that \( (\text{isu}(p, t), (p_0, t_0)) \in \text{HB}^1 \), \( ((p_1, t_1), \text{com}(p, t)) \in \text{HB}^1 \), and \( ((p_0, t_0), (p_1, t_1)) \in \text{HB} \). It is important to note that \( (p_0, t_0) \) and \( (p_1, t_1) \) might be identical in which case \( \beta = (p_0, t_0) \).

Notice that every event in \( \beta \) (including \( (p_0, t_0) \) and \( (p_1, t_1) \)) cannot write to a variable that \( (p, t) \) writes to under SI semantics, thus store order relation is not possible. Also, since \( (p, t) \) is not visible to any event in \( \beta \) thus the read-from and program order are not possible. Thus, the only possibility is that \( (\text{isu}(p, t), (p_0, t_0)) \in \text{RW} \), and \( ((p_1, t_1), \text{com}(p, t)) \in \text{RW} \).

C Proofs of Section 5. The Complete Instrumentation

In this section, we present the instrumentation for the remaining instructions which are, begin and commit for the attacker and happens-before helper.

C.1 Instrumentation of the Attacker

We provide in Figure 9 the instrumentation of the code for the attacker process. When the attacker randomly chooses a transaction to delay, it sets the flag \( a_{\text{tra}} \) to true in the instruction begin (line 15). Then, it sets the flag \( p.a \) to 1 to
indicate that the current process is the attacker. It copies the values from every variable \( x \) to its primed version \( x' \).

In the case the attacker starts the happens-before chain, it has to set the variable \( HB \) to \( true \) to mark the start of the happens-before chain and the end of the visibility chain and set the flag \( x.event \) to \( ld \) (line (2) in Figure 3). We can notice that when the \( HB \) is set to \( true \), we can no longer execute new transactions from the attacker (all conditions in lines (13) and (14) become \( false \)).

![Fig. 9: Instrumentation of the Attacker](image)

\[
\begin{align*}
[l_1]: \text{com; goto } l_2; & = \quad \text{// Typical execution of commit} \\
& l_1: \text{assume } a_{tr_A} = \bot \lor p.a \neq \bot; \text{ goto } l_{x1}; \\
& l_{x1}: \text{com; goto } l_2; \\
& \text{// Succeeded Commit of delayed transaction} \\
& l_1: \text{assume } a_{tr_A} \neq \bot \land HB \neq \bot; \text{ goto } l_{x2}; \\
& l_{x2}: \text{com; goto } l_2; \\
& \text{// Failed commit of delayed transaction} \\
& l_1: \text{assume } a_{tr_A} \neq \bot \land HB = \bot; \text{ assume false; } \quad (12)
\end{align*}
\]

\[
\begin{align*}
[l_1]: \text{begin; goto } l_2; & = \quad \text{// Typical execution of begin} \\
& l_1: \text{assume } HB = \bot \lor (p.a \neq \bot \lor a_{tr_A} = \bot); \text{ goto } l_{x1}; \quad (13) \\
& l_{x1}: \text{begin; goto } l_2; \\
& \text{// Begin of delayed transaction} \\
& l_1: \text{assume } HB = \bot \land a_{tr_A} = \bot; \text{ goto } l_{x2}; \quad (14) \\
& l_{x2}: \text{begin; goto } l_{x3}; \\
& l_{x3}: p.a := 1; \text{ goto } l_{x4}; \\
& l_{x4}: \text{Foreach } x \in \mathbb{V}. x' := x; \text{ goto } l_{x5}; \\
& l_{x5}: a_{tr_A} := \text{true; goto } l_2; \quad (15)
\end{align*}
\]

C.2 Instrumentation of the Happens-Before Helpers

In Figure 10 we provide the instrumentation of the remaining instructions of a happens-before helper.

When the flag \( HB \) is set to \( true \), a process (which cannot be the attacker, i.e., the flag \( p.a \) is null) starts the attempts to join the set of happens-before helpers. Thus, it randomly chooses a first transaction (the \( \text{begin} \) of this transaction is shown in line (16)) through which the process will join the set of happens-before helpers. When a process chose the transaction to join happens-before helpers, that means it has made pledge that during this transaction it will either do read from a variable that was updated by another delayed transaction from some other process in happens-before helpers or write to a variable that was accessed with a read or write from another process in happens-before helpers. When either one of these criteria are satisfied the flag \( p.hbh \) will be set to \( true \). If a process does not keep its pledge (the flag \( p.hbh \) is null) then before executing the \( \text{com} \) instruction of the first transaction we block the execution (line (20)).

The happens-before helpers processes continue executing their instructions, until one of them executes a load that reads from the shared variable \( x \) that was stored in \( a_{tr_A} \) which implies the existence of a happens-before cycle. When executing the instruction \( \text{com} \) at the end of every transaction, we have a conditional check to detect if we have a load or a write accessing the variable \( x \) (lines (17), (18), and (19)). When the check detects that the variable \( x \) was accessed, the
execution goes to the error state (line 18) to indicate that the execution has produced an anomaly and we denote the reached state of the instrumented program’s execution, the error state.

\[
\begin{align*}
\text{[l1: begin; goto l2;]_\text{HB}} &= \text{[l1: begin; goto l2;]_\text{HB}} \\
\text{// Begin before joining happens-before} &\text{// Commit before joining happens-before} \\
l_1: \text{assume HB} = \perp \land p.a = \perp ; \text{goto l}_2; &\quad l_1: \text{assume HB} = \perp \land p.a = \perp ; \text{goto l}_2; \\
l_{11}: \text{begin; goto l}_2; &\quad l_{11}: \text{com; goto l}_2; \\
\text{// Begin of first transactions to join} &\text{// Commit after joining happens-before} \\
l_1: \text{assume HB} \neq \perp \land p.hbh = p.a = \perp ; \text{goto l}_2; &\quad l_1: \text{assume HB} \neq \perp \land p.hbh \neq \perp ; \text{goto l}_2; \\
l_2: \text{begin; goto l}_2; &\quad (16) l_3: \text{f := a}_{\text{av}}; \text{goto l}_4; \\
\text{// Copying flags before the first transaction} &\text{// Commit after reaching the error state} \\
l_2: \text{begin; goto l}_3; &\quad (17) l_4: \text{f := f}.\text{event}; \text{goto l}_5; \\
\text{// starts executing and modifying the flags} &\text{// Commit after reaching the error state} \\
l_{12}: \text{Foreach } x \in \mathcal{V}, x.\text{event} := x.\text{event}; \text{goto l}_2; &\quad (18) l_{15}: \text{assume f} \neq \perp ; \text{assert false; } \\
\text{// Begin after joining happens-before} &\text{// Failed attempt to join happens-before} \\
l_1: \text{assume HB} \neq \perp \land p.hbh \neq \perp ; \text{goto l}_4; &\quad l_1: \text{assume HB} \neq \perp \land p.hbh = p.a = \perp ; \text{assume false; } (20)
\end{align*}
\]

Fig. 10: Instrumentation of Happens-Before Helpers Processes

As a direct consequence of Theorem 2, the next corollary states that some programs which have certain characteristics are robust against SI.

**Corollary 2.** Given a program $P$, if one of the following holds:

(a) every transaction of $P$ contains a single instruction either a read or a write;
(b) every transaction of $P$ contains only read/write events that access a single variable (different transactions might read/write to different variables);
(c) given a variable $x$, every transaction of $P$ contains a write to the variable $x$.

then $P$ is robust under SI.

**D Proofs of Section 5: Soundness and Completeness of the Instrumentation**

The aim of the instrumentation procedure is to reduce the problem of checking the existence of the anomaly described in Theorem 4 to reachability under serialisability of an error state by the instrumented version of a program. The instrumentation procedure is considered sound and complete iff if an error state is reachable, then we can reconstruct an anomaly, and every anomaly ensures that the error state is reachable by the instrumented version of the program.

**Theorem 5 (Soundness and Completeness).** A program $P$ is not robust iff the instrumented version of it, $P’$, reaches an error state under SER.
Proof. Soundness. Suppose that the instrumented program reaches an error state. Then, the execution’s trace of the instrumented program is of the form:
\[ \tau_* = \tau_1 \cdot \text{isu}(p, t) \cdot \tau_2 \cdot (p', t') \]
The last transaction, \((p', t')\) performed by a process \(p''\) that has a read accessing the variable \(x = \text{ata}\) and is part of the happens-before helpers. Because the conditional check can be performed only by a process \((p_{HB1})\) that is one of the happens-before helpers and is currently executing.

In order for \(p_{HB1}\), to join the set of happens-before helpers, it must have found that the valuation of the flag \(HB\) is not null which means there exists some process \(p\) that is the attacker that sets the flag \(HB\) to true. In \(\tau_1\), the attacker, happens-before helpers, and other processes start executing the original instructions without setting any flags or delaying any transactions. Afterwards, the attacker issues the delayed transaction \(\text{isu}(p, t)\) and it starts populating the primed variables \(x'\) and reading from them and setting the flags \(x.event'\) to 1 for every variable \(x\) that it writes to and \(y.event\) to \(ld\) for every variable \(y\) that it reads from. During the execution of \(t\), the attacker sets the flag \(HB\) to true. Hence, the happens-before helpers start checking at every instruction whether the flags \(x.event\) are set to either \(st\) or \(ld\). If so, they start populating the flags \(x.event\) and \(l.event\) as well. When \(HB\) is set to true, the attacker stop issuing new transactions. Therefore, all transaction in \(\tau_2\) are from the happens-before helpers.

We now transform \(\tau_*\) into the following execution trace:
\[ \tau = \tau'_1 \cdot \text{isu}(p, t) \cdot \tau'_2 \cdot \text{com}(p, t) \]
Here, \(\tau'_1\) is the subsequence of all \(\tau_1\) events that are produced by instructions from \(P\) without the conditionals checking (i.e., the assume statements). The transaction \(t\) which is executed by the attacker represents the delayed transactions in \(\tau\) with the removal of the conditionals checking and the flags setting. \(\tau'_2\) is the subsequence of all events of \(\tau_2\) produced by transactions from \(P\) which are executed only by the happens-before helpers except the conditionals checking and the flags setting. We add the commit of transaction \(\text{com}(p_0, t)\) to describe the commit of the delayed transaction that was delayed by the attacker. \(\tau\) is a possible execution’s trace of the program \(P\) because \(\tau_*\) is result of an execution of the instrumented version of \(P\) and we have removed from \(\tau\) all the effects of the instrumentation, and replaced the stores to auxiliary variables by issues of stores without changing the dependency between all the events in the execution.

All transactions in \(\tau'_2\) are from the happens-before helpers. Transactions in \(\tau'_2\) form a happens-before path between \(\text{isu}(p, t)\) and \(\text{com}(p, t)\). Also, we have \(a, b = (p', t') \in \tau'_2\) such that \((\text{isu}(p, t), a) \in \text{RW}(y)\) and \((b, \text{com}(p, t)) \in \text{RW}(x)\). No transaction in \(\tau'_2\) writes to a variable that \(t\) writes to. Hence, \(\tau\) indeed holds all the properties of the anomaly described in Theorem 4.

Completeness. Suppose we have an anomaly of a given program \(P\):
\[ \tau = \tau_1 \cdot \text{isu}(p, t) \cdot \tau_2 \cdot \text{com}(p, t) \]
such that $\tau$ maintains all the properties given in Theorem 4. We demonstrate that there is a possible serializable execution based on $\tau$ of the instrumented version of the program $P$ that reaches the error state. Next, we show how to build the instrumented program execution.

At the start of the execution, $\tau_1$, the attacker, happens-before helpers, and other processes execute the original transactions with just conditional checks. Afterwards, the attacker delays the transaction $isu(p, t)$ and starts populating the flags. In $isu(p, t)$, the attacker issues a store to the shared variable $'x' = a_{sta}$ and $\exists b \in \tau_2$ such that $(b, com(p, t)) \in RW(x)$. All writes that were executed in $t$ by the attacker are invisible to the remaining processes which includes the happens-before helpers. While executing $t$, the attacker sets the content of the flag $y.event$ to ld for every variable $y$ that it reads from and it sets the flag $HB$ to true.

On the other hand, the processes which are executing their transactions without delaying them will attempt to join the happens-before helpers by checking if the flag $HB$ is set to true. If so, they start the attempt of joining the happens-before helpers and when it succeeds they join the happens-before helpers and start executing their transactions which constitute $\tau_2$. The first executed transaction by the happens-before helpers is a described above which signals the start of $\tau_2$ and the happens-before dependency. Thus, in $\tau_2$, we have only transactions form the happens-before helpers (because the attacker stops when the flag $HB$ is set to true) such that they are related by the happens-before dependency that started from $isu(p, t)$ until it reaches $com(p, t)$ through $\tau_2$. We know that there must exist $b \in \tau_2$ such that $(b, com(p, t)) \in RW('x' = a_{sta})$. $b$ is equivalent to the last executed transaction by the happens-before helpers that accesses the shared variable $x$. Thus, the underlying happens-before helper will set the content of the flag $x.event$ to ld. Hence, when the underlying process executes the $com$ instruction of this transaction, it will go to the error state (lines 17, 18, and 19) and in this case the instrumented version of the program $P$ has reached the desired error state.

E Proofs of Section 6

The following theorem shows that the acyclicity of the commutativity dependency graph of a program implies the robustness of the program. Actually, the notion of robustness in this theorem relies on a slightly different notion of trace where store-order and write-order dependencies take into account values, i.e., store-order relates only transactions writing different values and the write-order relates a reading transaction to the oldest transaction (w.r.t. execution order) writing its value. In more details, we assume that two transactions are $WW$-related iff they write different values. Notice that since $RW$ is defined using $WW$ then when two transactions are $WW$-related iff they write different values then when two transactions are $RW$-related this implies that the value that is read is different than the one that is written. For example, in Figure 11a, if we don’t have this weakening of $WW$ we get an execution trace, where the happens-before
is cyclic, of the form $\tau = \text{isu}(p_1, t_1) \cdot (p_2, t_2) \cdot (p_3, t_3) \cdot \text{com}(p_1, t_1)$ because of the WW relation that links $(p_2, t_2)$ and $(p_3, t_3)$. However, with our weakening of WW there will be no WW relation between $(p_2, t_2)$ and $(p_3, t_3)$ and this the above execution can be equivalently rewritten as $\tau = (p_3, t_3) \cdot \text{isu}(p_1, t_1) \cdot \text{com}(p_1, t_1) \cdot (p_2, t_2)$ which is serializable. Similarly, we assume that two transactions $t_1$ and $t_2$ are related by WR iff when we swap the two transactions $t_2$ does not read the same value that $t_1$ is writing. For instance, in Figure 11 if we don’t have this assumption we get an execution trace, where the happens-before is cyclic, of the form $\tau = \text{isu}(p_1, t_1) \cdot (p_2, t_2) \cdot (p_3, t_3) \cdot \text{com}(p_1, t_1)$ because of the WR relation that links $(p_2, t_2)$ and $(p_3, t_3)$. However, with our assumption on WR there will be no WR relation between $(p_2, t_2)$ and $(p_3, t_3)$ and this the above execution can be equivalently rewritten as $\tau = (p_3, t_3) \cdot \text{isu}(p_1, t_1) \cdot \text{com}(p_1, t_1) \cdot (p_2, t_2)$ which is serializable. This approach helps in avoiding some of the harmless anomalies, where the happens before cycle might be caused by a write-write dependency between two transactions that writes the same values.

**Theorem 6.** For a program $P$, if (1) the commutativity dependency graph of $P$ does not contain non-mover cycles, then (2) $P$ is robust.

**Proof.** We prove the contrapositive, i.e., $\neg(2) \Rightarrow \neg(1)$. For the proof, we use the result of Theorem 2.

Assuming that the program $P$ is not robust. Then, based on Theorem 2, there must exist an execution of the instrumentation of $P$ that reaches the error state. We suppose that $t$ is the delayed transaction, $t_{ins}$ is the instrumentation of $t$ (writes are stored in auxiliary registers), and $p$ is the attacker process. Therefore, the execution of the instrumentation of $P$ that reaches the error state is of the form $\tau = \alpha \cdot (p, t_{ins}) \cdot a \cdot \beta \cdot b$ where $a$ writes to a variable that $t$ reads from and $b$ reads from a variable that $t$ writes to. We assume that $b$ is the first event that does read that accesses a variable that $t$ writes to. In the following we show that the commutativity dependency graph of $P$ contains a non-mover cycle where $t$ is $t_0$. We consider two cases, first case when $a = b$ and $\beta = \epsilon$, and second case is when $a \neq b$.

First case: $\tau = \alpha \cdot (p, t_{ins}) \cdot a$ where $a$ writes to a variable that $t$ reads from, reads from a variable that $t$ writes to, and does not write to a variable that $t$ writes to. Assume that $a = (p_1, t_1)$. Thus, we can safely obtain $\tau_0 = \alpha \cdot (p_1, t_1) \cdot (p, t')$ a serializable execution trace of $P$ where $t'$ is the reads free instantiation of $t$. Since $((p_1, t_1), \text{com}(p, t)) \in \text{RW}$ then $t_1$ reads a value that $t'$ is overwriting with a different value. Therefore, $\tau'_0 = \alpha \cdot (p, t') \cdot (p_1, t_1)$ is
either a serializable execution with a different end state than $\tau_0$ has or it is not an serializable execution. Thus, $(t_1, t') \in M_{RW}$ and $t_1$ does not write to a variable that $t$ writes to. Similarly, we can safely obtain $\tau_n = \alpha \cdot (p, t'') \cdot (p, t_1)$ a serializable execution trace of $P$ where $t''$ is the reads free instantiation of $t$. Since $(isu(p, t), (p, t_1)) \in RW$ then $t''$ reads a value that $t_1$ is overwriting with a different value. Therefore, $\tau'_n = \alpha \cdot (p_1, t_1) \cdot (p, t'')$ is either a serializable execution with a different end state than $\tau_n$ has or it is not an serializable execution. Thus, $(t'', t_1) \in M_{RW}$ and $t_1$ does not write to a variable that $t$ writes to.

Second case: $\tau = \alpha \cdot (p, t_{ins}) \cdot a \cdot \beta \cdot b$ where $a$ writes to a variable that $t$ reads from, $b$ reads from a variable that $t$ writes to, and every transaction in $a \cdot \beta \cdot b$ does not write to a variable that $t$ writes to. Assume that $a = (p_1, t_1)$ and $b = (p_n, t_n)$. Since the transactions in $(p_1, t_1) \cdot \beta \cdot (p_n, t_n)$ constitute the happens-before path in the execution trace $\tau$. Then, for every $(p_i, t_i)$, $(p_{i+1}, t_{i+1}) \in (p_1, t_1) \cdot \beta \cdot (p_n, t_n)$ we have that $((p_i, t_i), (p_{i+1}, t_{i+1})) \in (PO \cup WR \cup WW \cup RW)$. In the case $((p_i, t_i), (p_{i+1}, t_{i+1})) \in (WR \cup WW \cup RW)$, we can safely obtain $\tau_i = \alpha \cdot \gamma \cdot (p_i, t_i) \cdot (p_{i+1}, t_{i+1})$ which is a serializable execution trace of $P$ where $\gamma$ either empty (i.e., $\epsilon$) or $\gamma = (p_1, t_1) \cdots (p_{i-1}, t_{i-1})$. Since, $(p_i, t_i) \cdot (p_{i+1}, t_{i+1}) \in (WR \cup WW \cup RW)$, then swapping $t_i$ and $t_i + 1$ will result in either reordering of writes or write overwrites a read, or read obtains a different value. Therefore, $\tau'_i = \alpha \cdot \gamma \cdot (p_{i+1}, t_{i+1}) \cdot (p_i, t_i)$ is either a serializable execution trace with a different end state than $\tau_i$ has or it is not an serializable execution trace. Thus, $(t_i, t_{i+1}) \in M_{RW}$. Also, we have that $t_i$ and $t_{i+1}$ do not write to a variable that $t$ writes to. Similar to the first case, we can safely obtain $\tau_0 = \alpha \cdot (p_1, t_1) \cdot \beta \cdot (p_n, t_n) \cdot (p, t')$ a serializable execution trace of $P$ where $t'$ is the reads free instantiation of $t$. Since $((p_n, t_n), com(p, t)) \in RW$ then $t_n$ reads a value that $t'$ is overwriting with a different one. Therefore, $\tau'_0 = \alpha \cdot (p_1, t_1) \cdot \beta \cdot (p, t') \cdot (p_n, t_n)$ is either a serializable execution with a different end state than $\tau_0$ has or it is not an serializable execution trace. Thus, $(t_n, t') \in M_{RW}$ and $t_n$ does not write to a variable that $t$ writes to. Furthermore, we can safely obtain $\tau_n = \alpha \cdot (p, t'') \cdot (p_1, t_1)$ a serializable execution trace of $P$ where $t''$ is the writes free instantiation of $t$. Since $(isu(p, t), (p, t_1)) \in RW$ then $t''$ reads a value that $t_1$ is overwriting with a different one. Then, $\tau'_n = \alpha \cdot (p_1, t_1) \cdot (p, t'')$ is either a serializable execution trace with a different end state than $\tau_n$ has or it is not an serializable execution trace. Thus, $(t'', t_1) \in M_{RW}$ and $t_1$ does not write to a variable that $t$ writes to.

# F Experiments Appendix

In this section we describe the applications we used to evaluate our techniques. For every table in the original application program, we added a boolean annotation in our formalization of the table in Boogie, in order, to inspect whether a given record does exist in the table or not. For instance, if we consider the following table `Customer(CustomerId, CustomerName)`, in Boogie, we formalize the table as two maps: `CustomerIdAlive` which takes a `CustomerId` and returns `true` if there is a customer with underlying id and `false` otherwise, `CustomerTable` which takes a `CustomerId` and returns the corresponding `CustomerName`. Also,
in certain cases we formalize a Table as multiple maps. Below, we describe each application.

**Auction** [5]: Which has five transactions that manipulate three tables: BIDS, ITEMS, and USERS. Transaction RegBid is for placing a bid on an item. Transaction RegUser is for registration a user. Transaction ViewItem is for viewing the number of bids for an item. Transaction ViewUser is for looking at a user name. Transaction ViewUsers is for looking at all registered users.

**Cassieq-Core** [2]: A core unit of a distributed queue. It has eight transactions that manipulate a single table: USERACCOUNTS. Transaction AddNewAccount is for adding a new account in USERACCOUNTS. Transaction DeleteAnAccount is for deleting an account from USERACCOUNTS. Transaction AddNewKey is for adding a new key to an existing account in USERACCOUNTS. Transaction DeleteAKey is removing a key from an existing account in USERACCOUNTS. Transaction GetAnAccount is to check whether there exist an account with a given id in the table USERACCOUNTS. Transaction GetAccounts is to return all existing accounts in USERACCOUNTS. Transaction GetAccountKeys is for getting all the keys of certain account in USERACCOUNTS. Transaction GetAccountKey is to check whether a certain account does hold a certain key in USERACCOUNTS.

**Courseware** [13, 18]: Which has five transactions that manipulate three tables: STUDENT, COURSE, and ENROLED. Transaction RegisterStudent is for registering a new student in the table STUDENT. Transaction AddCourse is for adding a new course in the table COURSE. Transaction EnrollStudent is for enrolling a given registered student in a given course. Transaction RemoveCourse is for removing a given course from the table COURSE. Transaction QueryCourses is for inspecting courses in the table COURSE.

**Currency-Exchange** [3]: A trading service. It has six transactions that manipulate a single table: TRADES. Transaction SaveTrade is for registering a new trade. Transaction ViewListTrades is for viewing the trades that occurred before a given instance of time. Transaction ViewTrade is for inspecting a given trade. Transaction ViewTradeUser is for looking for a user who carried out a given trade. Transaction GetNbTrades is for inspecting the number of trades. Transaction GetTradeTime Stamp is for looking for the time stamp of a given trade.

**FusionTicket** [15]: Which has four transactions that manipulate a single table: EVENTS. Transaction AddEvent is adding new event in some given venue. Transaction ViewEvent is for looking at given a event and the number of tickets available at this event. Transaction Browse is for looking at events that are planned in some given venue. Transaction Purchase is for buying a ticket at a certain event.

---

2 https://github.com/paradoxical-io/cassieq

3 https://github.com/Haiyan2/Trade
Shopping-Cart [4]: An on-line shop service implemented over Cassandra. It has four transactions that query two tables: USERS and PRODUCTS. Transaction GetUser is for querying the existence of a user in the table USERS. Transaction GetProductsByCategory is for finding products that are in a given category. Transaction GetProductByUPC is for finding a product through its UPC. Transaction GetCategories is for finding the categories.

Playlist [5]: An on-line music service. It has fourteen transactions that manipulate three tables: USERS, TRACKS, and ARTISTS. Transaction AddTrack is for adding a new track in the table TRACKS. Transaction GetTrack is for getting a certain track from the table TRACKS. Transaction AddUser is for adding a new user in the table USERS. Transaction GetUser is for looking for a certain user in the table USERS. Transaction CreatePlaylist is for creating a playlist of certain user in the table USERS. Transaction ListArtistByLetter is for listing artists by their first letters of their names. Transaction ListSongsByArtist is for listing tracks produced by certain artist. Transaction ListSongsByGenre is for listing tracks of certain genre type. Transaction AddTrackToPlaylist is for adding an existing track (in TRACKS) to an existing user play list in the table USERS. Transaction DeleteTrackFromPlaylist is for removing a track from user play list in the table USERS. Transaction GetPlaylistForUser is for getting the contents of certain play list of certain user. Transaction GetPlaylistNames is for getting all the play lists of certain user. Transaction DeletePlaylistForUser is for deleting a certain user’s play list. Transaction DeleteUser is deleting a user from the table USERS.

RoomStore [6]: A messages bot service. It has five transactions that manipulate a single table: MESSAGES. Transaction AddMessage is for adding a new message to the table MESSAGES. Transaction GetLastMessage is for getting the messages of given user. Transaction GetMessages is for looking for messages that were added in a certain date. Transaction GetSpecificMessage is for getting specific message that was added in a certain date and time. Transaction GetTopicMessages is for getting messages that are of certain topic.

SmallBank [2]: Which has five transactions that manipulate three tables: ACCOUNT, SAVING, and CHECKING. Transaction Balance is for looking at both the saving and checking balances of a given user account. Transaction DepositChecking is for depositing a certain amount into the checking balance. Transaction TransactSaving is for depositing or withdrawing into/from the saving balance. Transaction Amalgamate (Amg) is for moving the saving and checking balances of an account to another account checking balance and resetting the saving and checking balances of the first account to zero. Transaction WriteCheck is for withdrawing from a given account’s checking balance.

TPC-C [23]: Which has five transactions that manipulate nine tables: WAREHOUSE, DISTRICT, STOCK, ITEMS, CUSTOMERS, HISTORY, ORDER,

---

[4] https://github.com/nikhilswagle/Shopping_Cart_Angular_Cassandra
[5] https://github.com/DataStaxDocs/playlist
[6] https://github.com/mebigfatguy/roomstore
NEWORDER, and ORDERLINE. Transaction NewOrder is for placing a new order on a set of items. Transaction Delivery is for delivering a withstanding order at certain warehouse. Transaction Payment is for a given customer paying a withstanding amount of credit. Transaction OrderStatus is for inspecting certain orders and the associated order lines. Transaction StockLevel is for inspecting stocks at certain warehouse and the withstanding orders at this warehouse.