IsoPredict: Dynamic Predictive Analysis for Detecting Unserializable Behaviors in Weakly Isolated Data Store Applications

CHUJUN GENG, Ohio State University, USA
SPYROS BLANAS, Ohio State University, USA
MICHAEL D. BOND, Ohio State University, USA
YANG WANG, Ohio State University, USA

Distributed data stores typically provide weak isolation levels, which are efficient but can lead to unserializable behaviors, which are hard for programmers to understand and often result in errors. This paper presents the first dynamic predictive analysis for data store applications under weak isolation levels, called *IsoPredict*. Given an observed *serializable* execution of a data store application, IsoPredict generates and solves SMT constraints to find an *unserializable* execution that is a feasible execution of the application. IsoPredict introduces novel techniques that handle divergent application behavior; solve mutually recursive sets of constraints; and balance coverage, precision, and performance. An evaluation on four transactional data store benchmarks shows that IsoPredict often predicts unserializable behaviors, 99% of which are feasible.

CCS Concepts: • **Software and its engineering** → **Software testing and debugging**.

Additional Key Words and Phrases: weak isolation levels, dynamic predictive analysis, data stores, transactions

ACM Reference Format:
Chujun Geng, Spyros Blanas, Michael D. Bond, and Yang Wang. 2024. IsoPredict: Dynamic Predictive Analysis for Detecting Unserializable Behaviors in Weakly Isolated Data Store Applications. *Proc. ACM Program. Lang.* 8, PLDI, Article 161 (June 2024), 25 pages. https://doi.org/10.1145/3656391

1 INTRODUCTION

Distributed data stores are the foundation of today's service infrastructure, due to their scalability, fault tolerance, and ease of use [16, 20, 38, 47]. Many real-world data stores only support *weak isolation* levels, such as *causal consistency* (CAUSAL) [3], which is the strongest level that achieves availability under network partitions [13, 28]. Another weak isolation level is *read committed* (RC) [5], which is commonly used by database applications to balance performance and correctness [15, 17, 40, 49]. Under weak isolation, an execution may be *unserializable*, producing an outcome that is impossible for any serial execution. Unserializable behaviors are poorly understood by most programmers, and often lead to errors and failures in real-world systems [15, 49, 51].

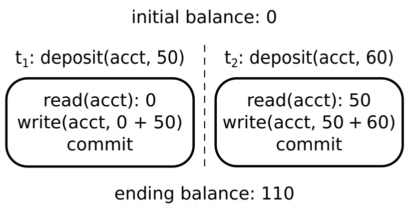
Prior work has introduced techniques to find unserializable behaviors in data store applications under weak isolation, but has scalability or accuracy limitations. Static analysis can find unserializable behaviors, but its precision scales poorly with program complexity, leading to many false

Authors' addresses: Chujun Geng, Ohio State University, Columbus, USA, geng.195@osu.edu; Spyros Blanas, Ohio State University, Columbus, USA, blanas.2@osu.edu; Michael D. Bond, Ohio State University, Columbus, USA, mikebond@cse.ohio-state.edu; Yang Wang, Ohio State University, Columbus, USA, wang.7564@osu.edu.

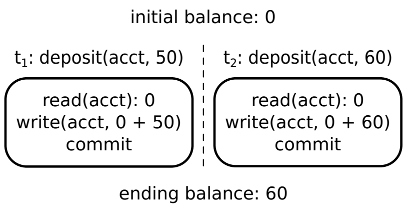
ACM 2475-1421/2024/6-ART161
https://doi.org/10.1145/3656391

Algorithm 1 A procedure in a data store application that deposits money in an account.

```
procedure DEPOSIT(account, amount)
    balance ⟵ DataStore.get(account)            ▷ Read balance; implicitly starts transaction if not in one
    DataStore.put(account, balance + amount)     ▷ Update balance
    DataStore.commit()                           ▷ Commits transaction
```



(a) The execution in which t_2 reads from t_1 is CAUSAL, RC, and SERIALIZABLE.



(b) The execution in which t_1 and t_2 both read from the initial state is CAUSAL and RC but not SERIALIZABLE.

Fig. 1. Different executions of two sessions (clients) concurrently on the same account.

positives (infeasible unserializable behaviors) [12, 39, 43]. Dynamic analysis can avoid false positives by analyzing only the observed execution [7, 11], or it can extrapolate from an observed execution but report numerous false positives [23, 51]. §8 discusses prior work in more detail.

Motivating example. Algorithm 1 shows code of a transactional data store application. The *DataStore* provides a key–value interface. Our execution model requires that every *get* (read) or *put* (write) operation to execute in a transaction, so an operation starts a new transaction if the current session (i.e., client) is not in a transaction. A *commit* operation ends the session's ongoing transaction.

Figure 1 shows two different executions of the application. In each execution, two sessions (i.e., clients) call DEPOSIT concurrently on the same empty account to deposit 50 and 60, respectively. Developers would expect that the ending balance will be 110, which is the only serializable outcome. However, under weak isolation levels CAUSAL and RC, the ending balance may be 110, 50, or 60.

Contributions. This paper introduces IsoPredict, the first predictive analysis for transactional data store applications, and shows that the approach is effective at finding unserializable behaviors. Given a serializable execution such as Figure 1a as input, IsoPredict finds an unserializable execution such as Figure 1b. IsoPredict uses *dynamic predictive analysis*, which analyzes an observed execution of a program and detects alternative *feasible, unserializable* executions of the program.

Predictive analysis is powerful because, in essence, it explores many executions at once. To predict an unserializable execution from an observed serializable execution, IsoPredict generates SMT constraints that encode execution feasibility, unserializability, and weak isolation level (CAUSAL or RC), and uses an off-the-shelf SMT solver to solve them. We introduce analysis variants that trade coverage for performance, and precision for coverage. To account for the possibility of predicting infeasible executions, IsoPredict can optionally *validate* a predicted unserializable execution. An evaluation on transactional data store benchmarks shows that IsoPredict is effective at predicting unserializable executions from observed executions under CAUSAL and RC. More than 99% of predictions are validated as feasible executions.

While prior work introduces predictive analysis for shared-memory programs [30, 33, 44–46, 50], to our knowledge IsoPredict is the first predictive analysis approach for transactional data store applications, which present unique challenges (§8). Compared to prior work MonkeyDB [7], IsoPredict is comparably effective at finding unserializable executions of the evaluated programs (§7.3).

However, IsoPredict and MonkeyDB use completely different approaches to find erroneous executions. While MonkeyDB uses random exploration to produce an erroneous execution, IsoPredict uses predictive analysis to evaluate an equivalence class of many executions at once. Furthermore, MonkeyDB requires applications to run on its specialized data store, while IsoPredict's predictive analysis approach is in principle suitable for analyzing executions from any data store, although demonstrating so is outside the scope of this paper.

2 BACKGROUND

This section introduces this paper's formalisms for weakly isolated executions of transactional data store applications, which are closely based on the axiomatic framework of Biswas and Enea [6]. We use this framework because it supports a variety of isolation levels, is well suited to encoding as constraints, and has been employed by recent work [7, 10].

2.1 Weakly Isolated Execution Histories

A transactional data store is modeled as a distributed store of key–value pairs. A data store application performs read (get) and write (put) operations on keys, all executed in transactions. Non-transactional applications can be handled by treating each read and write operation as a separate transaction. An execution consists of *events* in committed transactions (aborted transactions are not part of an execution). Each event is either $\mathsf{read}(k)$, or $\mathsf{write}(k)$ or commit, where k is a key. Other operations, such as insertion into and deletion from a set, can be modeled in terms of reads and writes. Multiple clients may open connections, or *sessions*, to the data store. If a session is not in a transaction, its next event implicitly starts a new transaction, ensuring every event is in a transaction. The commit event ends the current transaction. Within a session, transactions are ordered by the strict partial order *session order* (so):

$$so(t_1, t_2) \coloneqq t_1 \text{ precedes } t_2 \text{ in the same session}$$

An important property of an execution is *which* write each read reads from. The strict partial order wr_k (write–read on key k) orders transactions if one reads from the other:

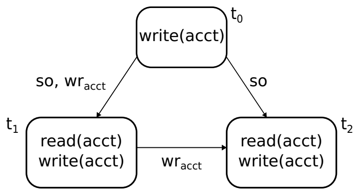
$$wr_k(t_1, t_2) \coloneqq t_2 \text{ reads the write of } t_1 \text{ on } k$$

If a read reads from a write in the same transaction, the read is not included as an event in the transaction (and thus this write–read ordering is not included in wr_k). If a transaction writes k multiple times, only the last write is included as an event in the transaction. Thus a $\mathsf{read}(k)$ event always reads from a $\mathsf{write}(k)$ in another transaction, which is the transaction's last write to k. If a transaction t reads k from the data store's initial state, then $wr_k(t_0, t)$, where t_0 is a special transaction representing the initial state. The union of wr_k over all keys is wr, i.e., $wr \coloneqq \bigcup_{k \text{ is a key}} wr_k$. The transitive closure of so and wr is *happens-before* order, i.e., $hb \coloneqq (so \cup wr)^+$. An execution *history* of a data store application is the set of all committed transactions (T), session order (so), and write–read order (wr), i.e., $History \coloneqq \langle T, so, wr \rangle$. Every history includes the special transaction t_0 mentioned above that represents the initial state. t_0 implicitly writes the initial value to every key, and t_0 is so-ordered before all other transactions.

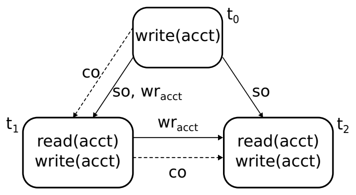
Example. Figures 2a and 3a each show an execution history as a graph. Transactions are boxes containing read and write events implicitly concluded by a commit event. t_1 and t_2 execute in different sessions, and t_0 is the initial state transaction. The wr_k edges indicate each read's writer.

2.2 Serializablility

An execution history $\langle T, so, wr \rangle$ is SERIALIZABLE if and only if it could have been produced by a serial execution of the transactions in T. (In a serial execution, transactions execute one at a time, and

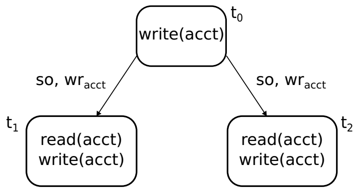


(a) Execution history

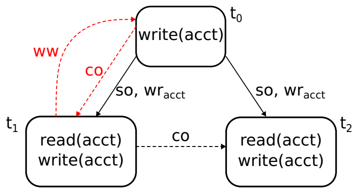


(b) A *co* (dashed arrows) consistent with the SERIALIZABLE axioms.

Fig. 2. A CAUSAL, **SERIALIZABLE** history corresponding to Figure 1a.



(a) Execution history



(b) A *co* (dashed arrows) inconsistent with the SERIALIZABLE axioms (contradiction shown in red).

Fig. 3. A CAUSAL, **UNSERIALIZABLE** history corresponding to Figure 1b.

every read to k reads from the most recent write to k.) Equivalently, an execution is SERIALIZABLE if and only if there exists a *commit order*, *co*, with the following constraints: (1) *co* must be consistent with happens-before (*hb*) order. (2) a transaction that writes to k cannot be *co*-ordered between two transactions ordered by wr_k. The second constraint's ordering is called *arbitration order* and represented by the strict partial order *ww*, which is defined as follows:

$$ww(t_1, t_2) := \exists k, t_1 \text{ and } t_2 \text{ write to } k \wedge \exists t_3 \in T, wr_k(t_2, t_3) \wedge co(t_1, t_3) \tag{1}$$

Note the circular dependency between *ww* and *co*: Commit ordering may imply additional arbitration ordering, which in turn may imply additional commit ordering. This property leads to challenges in encoding SMT constraints that §4 explains and addresses. Thus a history is SERIALIZABLE if and only if there exists a *co* that is consistent with *hb* and *ww*:

$$\langle T, so, wr\rangle \text{ is SERIALIZABLE} \iff \exists co, hb \cup ww \subseteq co$$

Equivalently, the history is SERIALIZABLE if and only if there exists *co* such that $(hb \cup ww \cup co)^+$ is acyclic. An execution is UNSERIALIZABLE if and only if it is *not* SERIALIZABLE.

Example. Figure 2a's history is SERIALIZABLE because there exists a commit order ($t_0 <_{co} t_1 <_{co} t_2$), shown in Figure 2b, that is consistent with the SERIALIZABLE axioms. Note that the arbitration rule (Equation 1) never applies in Figure 2a, and so Figure 2b shows no *ww* edges.

The history in Figure 3a is UNSERIALIZABLE because there does *not* exist a commit order that satifies the SERIALIZABLE axioms. For example, as Figure 3b shows, if $co(t_1, t_2)$, then $ww(t_1, t_0)$ by Equation 1, which implies $co(t_1, t_0)$ and thus *co* is cyclic. Alternatively, if $co(t_2, t_1)$, then $ww(t_2, t_0)$ and thus $co(t_2, t_0)$, and again *co* is cyclic.

2.3 Causal Consistency

Causal consistency (CAUSAL) is a weak isolation level that preserves the order of operations that are causally related [3]. CAUSAL is of theoretical and practical interest because it is the strongest isolation level achievable when a data store requires availability under network partitions [13, 28, 36].

Similar to SERIALIZABLE, CAUSAL is defined in terms of whether there exists a commit order that is consistent with happens-before (hb) and an arbitration order, which we call ww_{causal} to distinguish it from the arbitration order for SERIALIZABLE (ww). Two transactions t_1 and t_2 are ordered by ww_{causal} if they write the same key and if there is a third transaction t_3 that happens-after t_1 ($hb(t_1, t_3)$) and reads from t_2's write to the same key ($wr(t_2, t_3)$). More formally,

$$ww_{causal}(t_1, t_2) \coloneqq \exists k, t_1 \text{ and } t_2 \text{ write to } k \land \exists t_3 \in T, wr_k(t_2, t_3) \land hb(t_1, t_3) \tag{2}$$

A history is CAUSAL if and only if there exists a commit order consistent with hb and ww_{causal}:

$$\langle T, so, wr\rangle \text{ is CAUSAL } \iff \exists co, hb \cup ww_{causal} \subseteq co \tag{3}$$

Equivalently, a history is CAUSAL if and only if $(hb \cup ww_{causal})^+$ is acyclic.[1]

Example. The history in Figure 2a is CAUSAL because there exists a commit order $t_0 <_{co} t_1 <_{co} t_2$ that is consistent with the CAUSAL axioms. (Or, since the history is SERIALIZABLE, which is strictly stronger than CAUSAL, the history must be CAUSAL.) The history in Figure 3a is CAUSAL because there exists a commit order, $t_0 <_{co} t_1 <_{co} t_2$ (or $t_0 <_{co} t_2 <_{co} t_1$), that is consistent with the CAUSAL axioms.

2.4 Read Committed

Read committed (RC) is a popular weak isolation level because of the balance between performance and consistency it provides [5]. Whereas CAUSAL requires transactions ordered by happens-before (hb) to be viewed by other transactions in the same order, RC's arbitration order, ww_{rc}, only applies to write transactions that are read by multiple read events from the same transaction. More formally, RC is defined based on whether there exists a commit order that is consistent with hb and ww_{rc}, which is defined as follows:

$$ww_{rc}(t_1, t_2) \coloneqq \exists k, t_1 \text{ and } t_2 \text{ write to } k \land \exists \alpha, \beta, po(\beta, \alpha) \land \overline{wr}_k(t_2, \alpha) \land \exists k', \overline{wr}_{k'}(t_1, \beta) \tag{4}$$

where po is *program order*, a strict partial order that orders events within a transaction; and $\overline{wr}_k(t, e)$ is true if and only if e is a read *event* that reads from a write in transaction t (and thus $e \neq t$). Thus α and β must be events in the same transaction such that α is a read(k) event that reads from write(k) in t_2, and β is a read event that reads from any write in t_1. An execution history is RC if and only if there exists a commit order that is consistent with hb and ww_{rc}:

$$\langle T, so, wr\rangle \text{ is RC } \iff \exists co, hb \cup ww_{rc} \subset co \tag{5}$$

Example. The execution histories in Figures 2a and 3a are RC because there exist commit orders (in fact, the same commit orders used to establish CAUSAL) satisfying the above condition. Or, the histories are RC because they are CAUSAL, which is strictly stronger than RC.

3 ISOPREDICT OVERVIEW

IsoPredict consists of two main components, as shown in Figure 4: *predictive analysis* and *validation.*

The predictive analysis component takes as input an *observed* execution history that is recorded at the client application's backend data store, generates SMT constraints, and uses an SMT solver to find a predicted UNSERIALIZABLE execution if one exists. §4 describes IsoPredict's predictive analysis.

[1] Unlike SERIALIZABLE, CAUSAL can be defined in terms of whether $(hb \cup ww_{causal})^+$ is acyclic, which implies that a total commit order must exist. In contrast, SERIALIZABLE's arbitration order (ww) is dependent on the commit order, so SERIALIZABLE must be defined in terms of whether $(hb \cup ww \cup co)^+$ is acyclic.

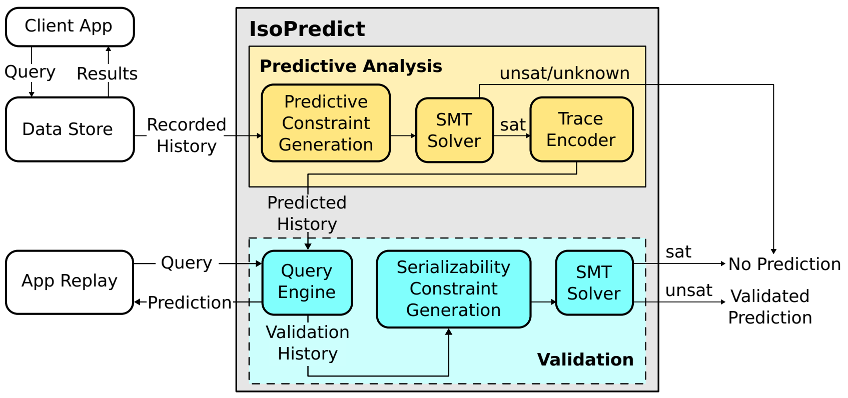


Fig. 4. IsoPredict's components and workflow.

The validation component tries to execute the predicted execution history to determine if it is feasible, and it generates and solves constraints to determine if the resulting execution is UNSERIALIZABLE. If so, IsoPredict outputs the validated history alongside a visualization of the validated UNSERIALIZABLE execution. §5 describes IsoPredict's validation component.

Validation is optional; developers may choose to skip it for two reasons. First, it may be overkill—in our experiments, over 99% of predicted UNSERIALIZABLE executions are successfully validated. Second, validation may be impractical if the application cannot be replayed easily. Validation is, however, useful to our evaluation to measure how many predicted executions are feasible.

4 PREDICTIVE ANALYSIS

IsoPredict's predictive analysis component takes as input an *observed* execution history of a data store application. The observed history $History = \langle T, so, wr_{obs} \rangle$ consists of a set of transactions T, session order so between transactions, and observed write–read ordering wr_{obs}. The goal of IsoPredict is to find a feasible, UNSERIALIZABLE execution that is valid under a weak isolation model M (i.e., CAUSAL or RC). To find such an execution, IsoPredict encodes and solves the following necessary and sufficient constraints for a *predicted* execution history, $History' = \langle T', so, wr \rangle$:

(1) $History'$ must be a feasible execution prefix[2] of the program that produced $History$ (§4.1).
(2) $History'$ must be UNSERIALIZABLE (§4.2).
(3) $History'$ must be valid under M (§4.3).

As an example, Figure 2a shows a SERIALIZABLE execution history that contains two deposit transactions (Algorithm 1) running concurrently. IsoPredict generates and solves the constraints sketched above, in order to predict the CAUSAL and RC but UNSERIALIZABLE execution from Figure 3a.

4.1 Encoding of Feasible Execution

This section describes the constraints that IsoPredict generates to ensure that $History' = \langle T', so, wr \rangle$ is a feasible execution of the application that produced $History = \langle T, so, wr_{obs} \rangle$.

Session order. The predicted execution must preserve the observed execution's session order (so). IsoPredict generates constraints over a Boolean SMT function $\phi_{so}(t_1, t_2)$ that takes two transactions as input; a transaction is an SMT data type representing the set of all executed transactions T. The

[2] We allow T' to be a subset of T to exclude transactions that may diverge from the observed execution (§4.5). An execution prefix is sufficient: If $History'$ exists, a full execution history exists that has $History'$ as a prefix and meets the criteria above.

analysis generates the following constraints to preserve the observed execution's *so*:

$$\forall t_1, t_2 \in T, t_1 \neq t_2, \quad \begin{cases} \boxed{\phi_{so}(t_1, t_2)} & \text{if } so(t_1, t_2) \\ \boxed{\neg\phi_{so}(t_1, t_2)} & \text{otherwise} \end{cases}$$

For clarity, SMT constraints generated by IsoPredict are $\boxed{\text{boxed}}$ throughput the paper. The way to understand the above is that, for every $t_1, t_2 \in T$ such that $t_1 \neq t_2$,[3] the analysis generates a constraint—either $\phi_{so}(t_1, t_2)$ or $\neg\phi_{so}(t_1, t_2)$ depending on whether the transactions are ordered by *so*.

Write–read order. Each read in the predicted execution can potentially read from any transaction that writes the same key.[4] To help reason about multiple reads in a transaction to the same key that have different writer transactions (and to help exclude potentially divergent events; §4.5), we introduce the notion of an event's *position*: In each session, events are numbered with monotonically increasing integers. To ensure each read has exactly one writer transaction in the predicted execution, IsoPredict introduces an SMT function $\phi_{choice}(s, i)$ that takes as input a session and the position of a read event in the session, and returns the writer transaction that the read reads from. Like transactions, sessions are a finite SMT data type representing the set of all sessions. (Note that $\phi_{choice}(s, i)$ is left undefined if i is not the position of a read event in s.) IsoPredict generates the following constraints to ensure that $\phi_{choice}(s, i)$ is equal to some transaction that writes the same key:

$$\forall k \text{ is a key}, \forall t_2 \text{ reads } k, \forall i \in rdpos_k(t_2), \quad \boxed{\bigvee_{t_1 \neq t_2 \text{ writes } k} \phi_{choice}(s_2, i) = t_1}$$

where s_2 is t_2's session, and $rdpos_k(t)$ is the set of positions of reads to k in transaction t.

IsoPredict encodes wr_k by generating constraints on Boolean SMT functions $\phi_{wr_k}(t_1, t_2)$:

$$\forall k \text{ is a key}, \forall t_1 \text{ writes } k, \forall t_2 \text{ reads } k, t_1 \neq t_2, \quad \boxed{\phi_{wr_k}(t_1, t_2) = \bigvee_{i \in rdpos_k(t_2)} \phi_{choice}(s_2, i) = t_1}$$

where s_2 is t_2's session.

To encode $wr(t_1, t_2)$, the analysis generates constraints on a Boolean SMT function $\phi_{wr}(t_1, t_2)$ that represents the union of all $\phi_{wr_k}(t_1, t_2)$:

$$\forall t_1, t_2 \in T, t_1 \neq t_2, \quad \boxed{\phi_{wr}(t_1, t_2) = \bigvee_{k \text{ is a key}} \phi_{wr_k}(t_1, t_2)}$$

4.2 Encoding Unserializability

This section describes how the analysis encodes constraints for the predicted execution to be UNSERIALIZABLE. The constraints must ensure that *all possible commit orders are cyclic.* §4.2.1 presents an approach that encodes the needed constraints exactly, resulting in long solving times. §4.2.2 presents an alternative approach that encodes a sufficient condition for unserializability, which has lower solving time than the first approach, but still has high coverage in our experiments.

[3] Although the partial and total orders throughout the paper are irreflexive, the analysis never needs to generate irreflexivity constraints (e.g., $\forall t, \boxed{\neg\phi_r(t, t)}$ for relation r) because it never generates any constraints that *use* $\phi_r(t, t)$.

[4] Recall that a read to k can only read from another transaction's *last* write to k (§2.1).

4.2.1 Constraints that encode an exact condition. To encode that no acyclic *co* exists for the predicted execution history, IsoPredict generates the following constraint:

$$\boxed{\forall \phi_{co}, \neg \mathit{IsSerializable}(\phi_{co})}$$

where *IsSerializable* is defined as shown below. Note that in the constraint above, $\phi_{co}(t)$, which takes a transaction t as input and evaluates to an integer indicating t's position in the *co* total order, is not an SMT function—it is a bound variable of the quantifier. Function *IsSerializable* is defined as follows:

$$\boxed{\begin{aligned} \mathit{IsSerializable}(\phi_{co}) \coloneqq {} & \mathit{Distinct}(\phi_{co}(t_1), \ldots, \phi_{co}(t_n)) \land \\ & \bigwedge_{\forall t_1, t_2 \in T, t_1 \neq t_2} (\phi_{wr}(t_1, t_2) \lor \phi_{so}(t_1, t_2) \lor \mathit{Arbitration}(t_1, t_2)) \Rightarrow \phi_{co}(t_1) < \phi_{co}(t_2) \end{aligned}}$$

where $t_1, \ldots, t_n$ are all transactions in T, and $\mathit{Distinct}(v_1, \ldots, v_k)$ is a built-in SMT function that requires all input values to be distinct from each other. By mapping ϕ_{co} (t) to a unique integer for each t, the first line of the equation above ensures that *co* is a total order.

The second line of the equation ensures that *co* is consistent with *wr*, *so*, and *ww*, respectively. For simplicity and to reduce the size of the constraints, arbitration constraints are factored out into the *Arbitration* function, which is defined as follows:

$$\boxed{\mathit{Arbitration}(t_1, t_2) \coloneqq \bigvee_{\substack{\forall k, t_1 \text{ and } t_2 \text{ write } k \\ \forall t_3 \in T \setminus \{t_1, t_2\}, t_3 \text{ reads } k}} \phi_{wr_k}(t_2, t_3) \land (\phi_{co}(t_1) < \phi_{co}(t_3))}$$

which is a straightforward encoding of the SERIALIZABLE arbitration constraints in Equation 1.

By using this approach we are pushing all the heavy lifting to the SMT solver. However, SMT solvers are known to be inefficient at solving constraints with universal quantifiers [34]—an issue confirmed by our performance results (§7.2).

4.2.2 Constraints encoding a sufficient but unnecessary condition. Alternatively, the analysis can encode a sufficient, but unnecessary, condition for predicting an UNSERIALIZABLE execution. We introduce a partial order, *pco*, that is a *subset of every commit order for every valid predicted execution.* If there exists a predicted execution for which *pco* is cyclic, then there cannot exist an acyclic *co* for the predicted execution, meaning it is UNSERIALIZABLE. In theory, this approach has the potential for missing UNSERIALIZABLE executions that §4.2.1's approach finds. But in our experiments, the *pco*-based approach predicts all UNSERIALIZABLE executions that §4.2.1's approach finds (§7.2).

We define *pco* to include all orders that must be in *co*: session (*so*), write–read (*wr*), and arbitration (*ww*) orders. We also introduce an *anti-dependency order* (*rw*) that must be in every *co*, which allows adding more edges to *pco* and thus finding more UNSERIALIZABLE executions. A challenge with encoding *pco* is that the arbitration and anti-dependency orders are both defined in terms of commit order, creating a circular dependency that leads to erroneous self-justifying edges in *pco.* We break both circular dependencies by introducing the notion of *rank* in the generated constraints. Next we describe anti-dependency order (*rw*), the circular dependency problem and our rank-based solution to it, and finally the constraints that the analysis generates.

Adding anti-dependency order (rw) to pco. To make *pco* as large as possible while still being consistent with every valid *co*, we add an *anti-dependency* (*rw*) order to *pco. rw* must be part of any valid *co*, as we prove in the extended version of this paper [25]. Intuitively, for any write–read relation $wr_k(t_1, t_2)$, anti-dependency prevents future transactions that also write k from being ordered between t_1 and t_2 in the commit order. More formally, we define $rw(t_1, t_2)$ as follows:

$$rw(t_1, t_2) \coloneqq \exists k, t_2 \text{ writes } k \land \exists t_w, wr_k(t_w, t_1) \land pco(t_w, t_2)$$

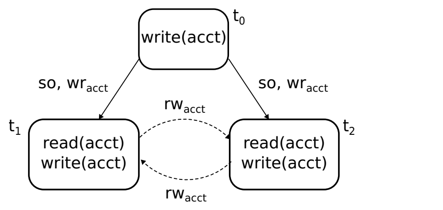


Fig. 5. Including anti-dependency ordering (*rw*; dashed arrows) in *pco* makes *pco* cyclic.

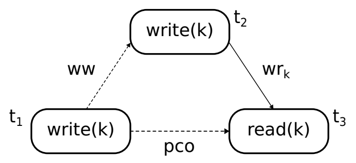


Fig. 6. An example of circular dependency: $ww(t_1, t_2)$ depends on $pco(t_1, t_3)$, which in turn depends on $ww(t_1, t_2)$.

Figure 5 shows an example in which *pco* is cyclic only if *rw* is included.

The partial order *pco* can now be defined as the union of all orders that must be part of *co*:

$$pco = (so \cup wr \cup ww \cup rw)^+$$

Adapting Equation 1 to use *pco* instead of *co*, we define arbitration order, *ww*, as follows:

$$ww(t_1, t_2) \coloneqq \exists k, t_1 \text{ and } t_2 \text{ write to } k \land \exists t_3 \in T, wr_k(t_2, t_3) \land pco(t_1, t_3)$$

Circular dependency and rank. In the definitions above, note the circular dependencies between *pco* and *ww* and between *pco* and *rw*, which seem to permit "self-justifying" edges. As an example, consider Figure 6. According to the definitions, $pco(t_1, t_3) \Rightarrow ww(t_1, t_2)$, and $ww(t_1, t_2) \Rightarrow pco(t_1, t_3)$, allowing us to *wrongly* conclude $ww(t_1, t_2)$ and $pco(t_1, t_3)$. To avoid such self-justifying edges, *pco*, *ww*, and *rw* in fact must be defined as the *minimal* relations that satisfy the above definitions.

How can we encode this "minimal relation" property in the SMT constraints? If IsoPredict simply encodes the above definitions as SMT constraints, the constraint solver will find self-justifying edges, resulting in spurious cycles and reporting executions that are not actually UNSERIALIZABLE. For example, for Figure 6, the SMT solver would choose both $ww(t_1, t_2)$ and $pco(t_1, t_3)$ to be true, finding a cycle and wrongly reporting a predicted execution that is actually SERIALIZABLE.

We address this problem by introducing the notion of *rank*, which orders *pco* edges that depend on each other. IsoPredict relies on an integer SMT function $rank(t_1, t_2)$ to enforce the following rule:

For any relations r *and* r'*, if* $r(t_1, t_2)$ *depends on* $r'(t'_1, t'_2)$*, then* $rank(t_1, t_2) > rank(t'_1, t'_2)$.

Note that the rule does not require $t_1 \neq t'_1$ or $t_2 \neq t'_2$. For Figure 6, rank constraints disallow $ww(t_1, t_2)$ and $pco(t_1, t_3)$, which would require both $rank(t_1, t_2) > rank(t_1, t_3)$ and $rank(t_1, t_3) > rank(t_1, t_2)$.

Generated constraints. IsoPredict generates arbitration and anti-dependency constraints on Boolean SMT functions $\phi_{ww}(t_1, t_2)$ and $\phi_{rw}(t_1, t_2)$:

$$\forall t_1, t_2 \in T, t_1 \neq t_2,$$

$$\boxed{\phi_{ww}(t_1, t_2) = \bigvee_{\substack{\forall k, t_1 \text{ and } t_2 \text{ write } k \\ \forall t_3 \in T \setminus \{t_1, t_2\}, t_3 \text{ reads } k}} \phi_{wr_k}(t_2, t_3) \land \phi_{pco}(t_1, t_3)) \land rank(t_1, t_2) > rank(t_1, t_3)}$$

$$\boxed{\phi_{rw}(t_1, t_2) = \bigvee_{\substack{\forall k, t_1 \text{ reads } k \land t_2 \text{ writes } k \\ \forall t_3 \in T \setminus \{t_1, t_2\}, t_3 \text{ writes } k}} \phi_{wr_k}(t_3, t_1) \land \phi_{pco}(t_3, t_2) \land rank(t_1, t_2) > rank(t_3, t_2)}$$

The following constraints ensure that *pco* is a partial order implied by *so*, *wr*, *ww*, and *rw*:

$$\forall t_1, t_2 \in T, t_1 \neq t_2,$$

$$\boxed{\begin{aligned}\phi_{pco}(t_1,t_2) = {} & \phi_{so}(t_1,t_2) \lor \phi_{wr}(t_1,t_2) \lor \phi_{ww}(t_1,t_2) \lor \phi_{rw}(t_1,t_2) \lor \\ & \bigvee_{t \in T \setminus \{t_1,t_2\}} \phi_{pco}(t_1,t) \land \phi_{pco}(t,t_2) \land rank(t_1,t_2) > rank(t_1,t) \land rank(t_1,t_2) > rank(t,t_2)\end{aligned}}$$

To ensure that *pco* is cyclic, the analysis generates the following constraint:

$$\boxed{\bigvee_{\forall t_1,t_2 \in T, t_1 \neq t_2} \phi_{pco}(t_1,t_2) \land \phi_{pco}(t_2,t_1)}$$

If the solver finds a satisfying solution, a predicted UNSERIALIZABLE execution exists. If the solver reports no satisfying solution, a predicted UNSERIALIZABLE execution may or may not exist. In our experiments, a predicted UNSERIALIZABLE execution never exists in this case.

We have not been able to come up with an execution for which our *pco*-based approach misses a predicted UNSERIALIZABLE execution. We believe that such an execution should exist because otherwise it would imply a polynomial-time algorithm for deciding if an execution history is serializable—a problem that is NP-hard [6].

4.3 Encoding Weak Isolation

This section describes the constraints that IsoPredict generates to ensure that the execution conforms to the target weak isolation model (CAUSAL or RC).

Regardless of the model, IsoPredict encodes *hb* as the transitive closure of *so* and *wr* (§2.1), by generating constraints on a Boolean SMT function $\phi_{hb}(t_1, t_2)$:

$$\forall t_1, t_2 \in T, t_1 \neq t_2, \quad \boxed{\phi_{hb}(t_1,t_2) = \phi_{so}(t_1,t_2) \lor \phi_{wr}(t_1,t_2) \lor \bigvee_{\forall t \in T \setminus \{t_1,t_2\}} \phi_{hb}(t_1,t) \land \phi_{hb}(t,t_2)}$$

4.3.1 Causal consistency (CAUSAL). To ensure that the predicted execution is CAUSAL, IsoPredict generates constraints that ensure that the transitive closure of CAUSAL arbitration order (ww_{causal}) and happens-before (*hb*) is acyclic (§2.3). IsoPredict encodes the CAUSAL axiom (Equation 2) by generating constraints on a Boolean SMT function $\phi_{ww_{causal}}(t_1, t_2)$ representing ww_{causal}:

$$\forall t_1, t_2 \in T, t_1 \neq t_2, \quad \boxed{\phi_{ww_{causal}}(t_1,t_2) = \bigvee_{\substack{\forall k, t_1 \text{ and } t_2 \text{ write } k \\ \forall t_3 \in T \setminus \{t_1,t_2\}, t_3 \text{ reads } k}} \phi_{wr_k}(t_2,t_3) \land \phi_{hb}(t_1,t_3)}$$

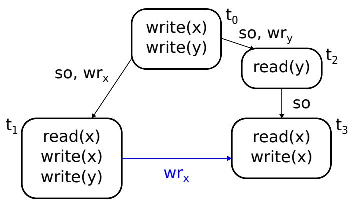
To ensure the execution is CAUSAL, there must exist a strict total order that is consistent with $(hb \cup ww_{causal})^+$ (Equation 3). IsoPredict generates the constraints on an *integer* SMT function $\phi_{co_{causal}}(t)$:

$$\forall t, t_1, t_2 \in T, t_1 \neq t_2, \quad \boxed{\phi_{hb}(t_1,t_2) \lor \phi_{ww_{causal}}(t_1,t_2) \implies \phi_{co_{causal}}(t_1) < \phi_{co_{causal}}(t_2)}$$

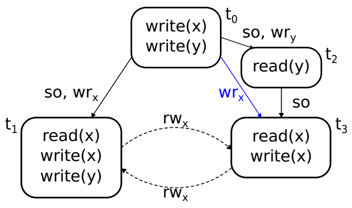
4.3.2 Read committed (RC). Similar to CAUSAL, IsoPredict generates constraints so that the transitive closure of RC arbitration order (ww_{rc}) and happens-before (*hb*) is acyclic (§2.4). IsoPredict encodes the RC axiom (Equation 4) with the help of a Boolean SMT function $\phi_{ww_{rc}}(t_1, t_2)$ that represents ww_{rc}:

$$\forall t_1, t_2 \in T, t_1 \neq t_2, \quad \boxed{\phi_{ww_{rc}}(t_1,t_2) = \bigvee_{\substack{\forall k,\ t_1 \text{ and } t_2 \text{ write } k \\ \forall t_3 \in T \setminus \{t_1,t_2\},\ t_3 \text{ reads } k \\ \forall i \in rdpos_*(t_3), \forall j \in rdpos_k(t_3),\ i<j}} \phi_{choice}(s_3,i) = t_1 \land \phi_{choice}(s_3,j) = t_2}$$

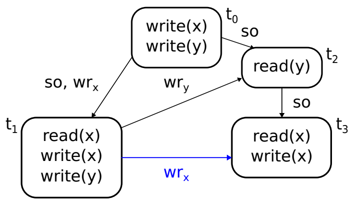
where $rdpos_*(t)$ is the set of positions of read events in transaction t, $rdpos_k(t)$ is the set of positions of read to k in transaction t, and s_3 is t_3's transaction. To ensure there exists a strict total order that



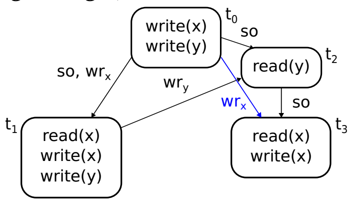
(a) An observed execution of Wikipedia for which a predicted CAUSAL, UNSERIALIZABLE execution exists.



(b) A CAUSAL, UNSERIALIZABLE prediction; the *pco* cycle (including *rw* edges) shows it is UNSERIALIZABLE.



(c) An observed execution of Wikipedia, for which *no* predicted CAUSAL, UNSERIALIZABLE execution exists.



(d) The non-CAUSAL execution that results if we try to change (c) so t_3 reads from t_0.

Fig. 7. Comparison of (relevant subsets of) executions from Wikipedia. Blue edges highlight the differences between observed and predicted executions.

is consistent with $(hb \cup ww_{rc})^+$ (Equation 5), IsoPredict generates constraints on an integer SMT function $\phi_{co_{rc}}(t)$:

$$\forall t, t_1, t_2 \in T, t_1 \neq t_2, \quad \boxed{\phi_{hb}(t_1, t_2) \vee \phi_{ww_{rc}}(t_1, t_2) \implies \phi_{co_{rc}}(t_1) < \phi_{co_{rc}}(t_2)}$$

4.4 Prediction Examples

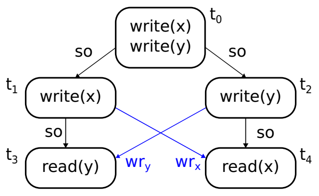
This section shows CAUSAL, UNSERIALIZABLE behaviors predicted by IsoPredict on programs evaluated in §7. The actual executions consist of dozens of transactions and thousands of events; the figures show only the transactions and events relevant to predicting UNSERIALIZABLE behavior.

Figure 7a shows an observed execution of the Wikipedia benchmark, and Figure 7b shows the CAUSAL, UNSERIALIZABLE execution predicted by IsoPredict. In contrast, Figure 7c shows a different observed execution of Wikipedia, from which no CAUSAL, UNSERIALIZABLE execution can be predicted. Figure 7d serves to illustrate that changing t_3's read of x to read from t_0 would lead to a non-*CAUSAL* execution (and thus will not be reported by IsoPredict).

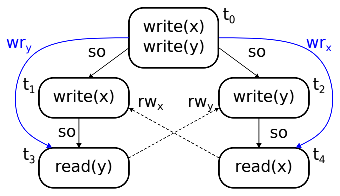
Figure 8a shows an observed execution of the Smallbank benchmark, and Figure 8b shows the IsoPredict-predicted execution. As Figure 8b shows, a CAUSAL, UNSERIALIZABLE predicted execution exists in which both reads read from the initial state (t_0), as demonstrated by the *pco* cycle $t_1 <_{co} t_3 <_{co} t_2 <_{co} t_4 <_{co} t_1$.

4.5 Handling Divergence in the Predicted Execution

Reading from a different write in the predicted execution than in the observed execution, may lead to different application behaviors. Specifically, code in the data store application that is *control dependent* on a read from a different writer transaction may generate different events. For example, consider the observed execution shown in Figures 9a and 9b, which executes transactions shown in Algorithms 1 and 2. Figure 9c shows an UNSERIALIZABLE predicted history that IsoPredict would find using the constraints presented so far. However, the predicted execution is infeasible: t_2 aborts



(a) An observed execution for which a CAUSAL, UNSERIALIZABLE predicted execution exists.



(b) A CAUSAL, UNSERIALIZABLE predicted execution as shown by the *pco* cycles including *rw* edges.

Fig. 8. Observed and predicted executions of Smallbank. For simplicity, each history shows a subset of the executed transactions, and each transaction shows a subset of the executed events.

Algorithm 2 A procedure in a data store application that withdraws money from an account.

```
procedure WITHDRAW($account$, $amount$)
    $balance \leftarrow DataStore.get(account)$            ▷ Read balance; implicitly starts transaction
    if $balance < amount$ then
        $DataStore.rollback()$                               ▷ Abort transaction
    else
        $DataStore.put(account, balance - amount)$           ▷ Update balance
        $DataStore.commit()$                                 ▷ Commit transaction
```

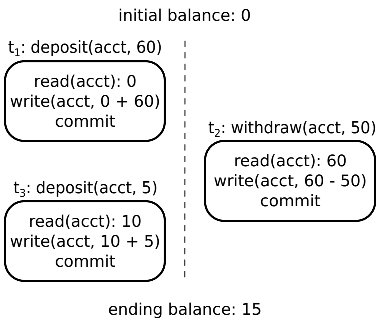
if it reads from t_0, making it impossible for t_3 to read from t_2, as Figure 9d shows. IsoPredict (mostly) avoids make spurious predictions, by excluding (much of the) potentially divergent behavior.

Divergent behavior. To account for divergent behavior, we make a distinction between the *predicted execution*, which is generated by IsoPredict based on the observed execution, and what we call the *validating execution*, which is the execution that actually occurs if one tries to produce the predicted execution using the data store application. Divergent behaviors are behaviors that differ between the predicted and validating executions. We categorize divergent behaviors into two categories:

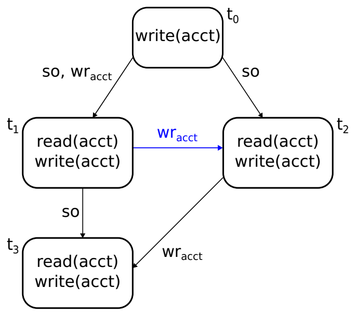
- The validating execution reads or writes different keys or omits or adds events from the predicted execution, leading to a different execution history with different properties.
- A transaction that commits in the predicted execution, aborts in the validating execution (e.g., an application might have logic that aborts if a consistency check fails), as Figures 9c and 9d show.

The problem with divergent behavior is that an UNSERIALIZABLE predicted execution can lead to a *SERIALIZABLE* validating execution. (The validating execution will always be a feasible execution conforming to the weak isolation model because validation ensures these properties; §5.)

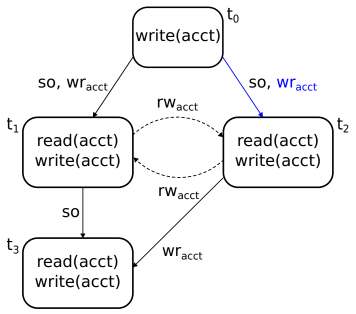
Prediction boundary. IsoPredict accounts for divergence by generating *prediction boundary* constraints that exclude events that may be impacted by divergence—specifically, events that *happen-after* (i.e., inverse of *hb*) any read event that reads from different writers in the predicted and observed executions. IsoPredict supports a prediction boundary that is *strict* or *relaxed*, as shown in Table 1. The *strict* boundary excludes events that happen-after *events* that read from a different writer in the predicted execution than in the observed execution. The strict boundary prevents false predictions except when a transaction in the predicted execution aborts in the validating execution. Alternatively, the *relaxed* boundary excludes events that happen-after *transactions* that read from a different writer, risking more false predictions but increasing the chances of finding an UNSERIALIZABLE predicted execution.



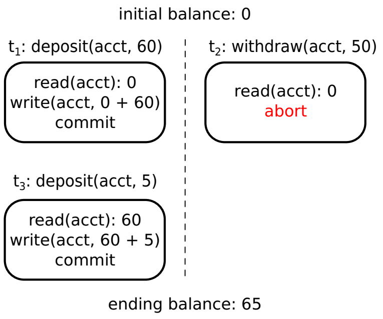
(a) One session deposits into an account twice, while another session withdraws once.



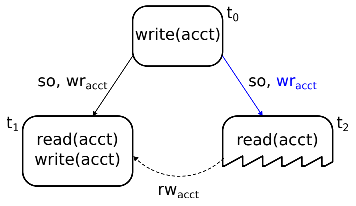
(b) The execution history for (a), which is SERIALIZABLE. The write–read edge from t_1 to t_2 (shown in blue) is not present in the predicted execution in (c).



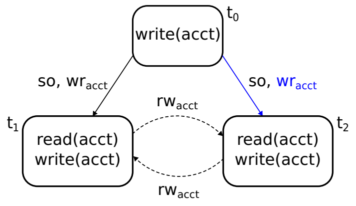
(c) A predicted execution history that is UNSERIALIZABLE. The write–read edge from t_0 to t_2 (shown in blue) was not present in the observed execution (b).



(d) The *validating* execution based on the predicted execution in (c). It diverges because t_2 aborts, and the resulting execution is SERIALIZABLE.



(e) This execution history consisting of the events from the predicted execution in (c) that are within the *strict* prediction boundary is SERIALIZABLE.



(f) This execution history consisting of the events from the predicted execution in (c) that are within the *relaxed* prediction boundary is UNSERIALIZABLE.

Fig. 9. Motivation for a prediction boundary (a–d) and illustration of the two kinds of prediction boundaries (e–f). The target weak isolation model is CAUSAL. Dashed arrows represent *pco* edges that are not part of the history.

Table 1. Comparison of strict and relaxed prediction boundaries.

Prediction boundary	**Excluded events**	**Divergent behaviors can cause false predictions**
Strict	Events that happen-after any *read event* with a different writer	Abort-related only
Relaxed	Events that happen-after any *transaction* containing a read event with a different writer	Any

Figures 9e and 9f show strict and relaxed boundaries, respectively, applied to the prediction in Figure 9c. The strict boundary excludes all *events* that happen-after t_2's read (since it has a different writer than in Figure 9b); the resulting execution history is SERIALIZABLE. The relaxed boundary excludes all *transactions* that happen-after t_2's read; the resulting execution history is UNSERIALIZABLE. Although the relaxed boundary allows a false prediction in this example, in our evaluation the relaxed boundary results in few false predictions.

Generating prediction boundary constraints. Here we present IsoPredict's constraints for excluding events using the prediction boundary. We show constraints for the *strict* prediction boundary, but the constraints for the *relaxed* prediction boundary are similar except they also constrain every session's boundary to be the last event of a transaction.

The prediction boundary is delimited by a *boundary event* in each session, which is either (1) a read event, which reads from a different write in the predicted execution than in the observed execution, or (2) the last event in the session (which will always be a commit event). IsoPredict generates the following constraints on an integer SMT function $\phi_{boundary}(s)$ to ensure that the boundary event for each session is either a read event or the last event (represented by position ∞):

$$\forall s \text{ is a session,} \quad \boxed{\left(\bigvee_{\substack{t \text{ is a transaction in } s\\ i \in rdpos_k(t)}} \phi_{boundary}(s) = i\right) \vee \phi_{boundary}(s) = \infty}$$

Recall that $rdpos_k$(s) is the set of positions of reads to k in the transaction t.

To ensure that each read that happens-*before* the prediction boundary reads from the same write as in the observed execution, IsoPredict generates the following constraints, where $\phi_{obs}(s, i)$ is an integer SMT function that represents the last write of each read in the *observed* execution history (and is thus the analogue of ϕ_{choice} for the observed execution):

$$\forall t_1, t_2 \in T, t_1 \neq t_2, \forall i \in rdpos_k(t_2) = i, t_2\text{'s read at pos } i \text{ reads from } t_1 \text{ in } wr_{obs}, \quad \boxed{\phi_{obs}(s_2, i) = t_1}$$

$$\forall k \text{ is a key}, \forall t_1 \text{ writes } k, \forall t_2 \text{ reads } k, \forall i \in rdpos_k(t_2), \quad \boxed{i < \phi_{boundary}(s_2) \implies \phi_{choice}(s_2, i) = \phi_{obs}(s_2, i)}$$

where s_1 is t_1's session and s_2 is t_2's session.

A read to k *on or before* the prediction boundary must read from a write to k that is *before* the prediction boundary. IsoPredict ensures this property by generating the following constraints:

$$\forall k \text{ is a key}, \forall t_1 \text{ writes } k, \forall t_2 \text{ reads } k, \forall i \in rdpos_k(t_2),$$

$$\boxed{\phi_{choice}(s_2, i) = t_1 \wedge i \leq \phi_{boundary}(s_2) \implies wrpos_k(t_1) < \phi_{boundary}(s_1)}$$

where s_1 is t_1's session, s_2 is t_2's session, and $wrpos_k$(t) is the position of t's last write to key k.

To exclude events after the prediction boundary, IsoPredict generates modified constraints for all arbitration and anti-dependency rules, as detailed in the extended version of this paper [25].

5 VALIDATION

Even by using the prediction boundary, IsoPredict's predictive analysis may report UNSERIALIZABLE predicted executions for which the corresponding validating execution is SERIALIZABLE. To rule out such predictions, IsoPredict can attempt to *validate* predicted executions, by executing the data store application based on the predicted execution history, and checking whether the resulting *validating execution* is UNSERIALIZABLE.

Validating execution. Validation produces the validating execution using a query engine that takes the predicted execution as input. At each read(k) event, the query engine checks that (1) the corresponding read in the predicted execution also read from k; (2) the writer transaction t from the predicted execution also wrote to k in the validating execution; and (3) reading from t in the validating execution will satisfy the weak isolation model (CAUSAL or RC). If any of these conditions is violated, we categorize the execution as having *diverged*, and the query engine chooses a different, weak isolation model–conforming writer for the read to read from. Note that it is always possible to keep executing while preserving CAUSAL or RC [10]. Furthermore, the validating execution may still be UNSERIALIZABLE, as our evaluation shows.

Recall that the predicted execution history contains events only up to the prediction boundary. To avoid serendipitously introducing UNSERIALIZABLE behaviors that were not part of the predicted execution (which could make it tough to measure the effectiveness of IsoPredict's predictive analysis), validation executes each transaction in full that is on the boundary or that happens-before any transaction on the boundary—and then it terminates the execution. This approach is sufficient: If this execution prefix is UNSERIALIZABLE, then so is the full execution.

Note that validation must directly control what transaction each read reads from, i.e., the write–read relation (*wr*). Our evaluation extends MonkeyDB [7] to allow explicit control of *wr* (§6). In settings where MonkeyDB cannot be used, such as production systems, there are other ways to control *wr*. One is using resource locks (e.g., sp_getapplock in SQL Server) to force specific transaction orders that produce the desired *wr* relation.

Checking serializability. Validation generates constraints to check whether the validating execution history is SERIALIZABLE (which can be encoded more efficiently than UNSERIALIZABLE, since SERIALIZABLE implies a total commit order exists). If the solver returns "satisfiable," IsoPredict reports no prediction. Otherwise (the solver returns "unsatisfiable"), IsoPredict reports the validating execution, which is known to be a feasible, UNSERIALIZABLE, weak isolation model–conforming execution.

6 IMPLEMENTATION

This section describes the implementation of IsoPredict, which is publicly available [26].

Predictive analysis. We implemented IsoPredict's predictive analysis (§4) as a Python program that uses Z3Py, the Python binding of the Z3 SMT solver [18]. Observed and predicted execution histories are in the form of traces containing read and write events and transaction and session identifiers, including the transaction that each read reads from. If Z3 finds a predicted UNSERIALIZABLE execution, it either reports the predicted execution history in both textual and graphical forms, or passes the predicted history to the validation component, depending on how IsoPredict is configured.

To generate observed execution traces, we extended the implementation of *MonkeyDB*, a transactional key–value data store [7]. MonkeyDB handles relational queries by translating them to key–value queries. MonkeyDB executes transactions serially, and we configured it to choose the latest writer at each read, so observed executions are always SERIALIZABLE.

Validation. IsoPredict's validation component replays the client application on a customized query engine that we also built on top of MonkeyDB. The query engine executes transactions one at a time, in an order dictated by the predicted execution, to ensure that read events always occur after their writers. At each read, the query engine chooses a last writer that satisfies the weak isolation model and, if possible, matches the predicted execution (§5). Validation uses Z3Py to generate and solve SMT constraints to determine if the validating execution history is UNSERIALIZABLE, reporting the validating execution to the user in both textual and graphical forms if so.

Table 2. IsoPredict prediction strategies.

Pred. strategy	**Encoding precision**	**Pred. boundary**	**Divergence ⇒ false predictions?**
Exact-Strict	Exact encoding	Strict	Only because of aborts
Approx-Strict	Approximate encoding	Strict	Only because of aborts
Approx-Relaxed	Approximate encoding	Relaxed	Yes

Algorithm 3 Code executed by each of Voter's transactions.

```
procedure VOTE(id)
    $votes \leftarrow DataStore.get(id)$
    if $votes < 1$ then
        $DataStore.put(id, 1)$
    $DataStore.commit()$
```

The customized query engine handles transaction aborts by rewinding the predicted execution trace to the beginning of the current transaction. In our experiments, every transaction that aborted during the observed execution also aborts during the validating execution—except in a few cases, when a transaction that aborted in the observed execution and *immediately precedes a committed transaction on the prediction boundary*, actually commits in the validating execution. As for other divergent behavior, the resulting validating execution may or may not be UNSERIALIZABLE.

7 EVALUATION

This section evaluates how effectively and efficiently IsoPredict predicts UNSERIALIZABLE executions under CAUSAL and RC, and it compares empirically against prior work MonkeyDB [7].

7.1 Methodology

Prediction strategies. Table 2 shows the combinations of unserializability constraints and prediction boundaries that we evaluated, which we call *prediction strategies*. The Exact-Strict prediction strategy uses precise encoding of unserializability (§4.2.1), while Approx-Strict and Approx-Relaxed encode the sufficient condition for unserializability (§4.2.2). Exact-Strict and Approx-Strict encode the strict prediction boundary, while Approx-Relaxed encode the relaxed prediction boundary.

Benchmarks. We evaluated IsoPredict and MonkeyDB using transactional workloads from *OLTP-Bench*, a database testing framework that generates various workloads for benchmarking relational databases [19]. Table 3 shows quantitative characteristics of the evaluated Benchmarks.

Our experiments used versions of the OLTP-Bench programs that the MonkeyDB authors ported to use simplified SQL queries recognized by MonkeyDB [7]. In these versions, each benchmark runs a nondeterministic number of transactions based on a specified time limit. For the purposes of our evaluation, we modified the benchmarks to be more deterministic for two reasons. First, determinism provides a more stable comparison among IsoPredict's prediction strategies. Second, determinism helps with validation, since the validating execution can run the benchmark with the same RNG seed that the observed execution used. (To use validation in a production setting, one should record and replay the application [22, 35].) We modified the benchmarks to be more deterministic by (1) fixing the number of sessions and transactions per session and (2) adding a random number generator (RNG) seed as a parameter to each benchmark. Although these modifications increase determinism, the benchmarks still execute nondeterministically because the *interleaving* of transactions is timing dependent. This source of nondeterminism does not hinder validation, which executes transactions in an order consistent with the predicted execution's *hb* relation.

Table 3. Average number of events and committed transactions across 10 trials of each OLTP-Bench program.

	Small workload				**Large workload**			
	KV accesses		**Committed txns**		**KV accesses**		**Committed txns**	
Program	Reads	Writes	Total	(Read-only)	Reads	Writes	Total	(Read-only)
Smallbank	669.7	14.7	11.0	(3.5)	1271.3	30.5	20.3	(6.6)
Voter	763.0	6.0	12.0	(11.0)	919.0	6.0	24.0	(23.0)
TPC-C	3297.3	763.0	11.9	(0.9)	7025.6	1502.4	23.8	(1.7)
Wikipedia	1067.7	55.1	9.9	(8.8)	2677.1	111.1	22.8	(20.6)

We configured each benchmark with both *small* and *large* workloads, in which three sessions each execute four or eight transactions, resulting in 12 or 24 *attempted* transactions, respectively. The number of *committed* transactions is somewhat fewer because all programs except Voter occasionally abort a transaction based on application-specific logic.

Platform. All experiments ran on an Intel Xeon server at 2.3 GHz with 16 cores, hyperthreading enabled, and 187 GB of RAM, running Linux.

7.2 IsoPredict's Effectiveness and Performance

Tables 4 and 5 show IsoPredict's effectiveness and performance at predicting UNSERIALIZABLE executions under CAUSAL and RC, respectively. For each benchmark and each of IsoPredict's three prediction strategies, we ran IsoPredict on 10 executions, each of which used one of 10 RNG seeds, which we kept consistent across prediction strategies and isolation levels.

Predictive analysis. The *Sat* column under *Prediction* reports the number of UNSERIALIZABLE executions (out of 10) that IsoPredict found. The Approx-Relaxed prediction strategy generally predicts more than the other strategies because it uses the relaxed boundary. Although Exact-Strict can theoretically predict more executions than Approx-Strict, this never happened in our experiments.

IsoPredict consistently predicts more UNSERIALIZABLE executions under RC than under CAUSAL, which makes sense because RC is strictly weaker than CAUSAL. Voter has the biggest difference—there were no successful predictions under CAUSAL. This is because every observed execution of Voter has only one writing (i.e., non-read-only) transaction (see Algorithm 3), which is not sufficient to predict an UNSERIALIZABLE execution under CAUSAL.[5] Similarly, IsoPredict has low prediction rates for Wikipedia, which has few writing transactions. In contrast, under RC, a transaction may legally read both the initial state and the writing transaction, which is why IsoPredict has higher prediction rates for Voter and Wikipedia under RC than under CAUSAL. §4.4 and the extended version of this paper [25] present some observed and predicted executions from the evaluated benchmarks.

Validation. We configured IsoPredict to validate every predicted UNSERIALIZABLE execution. The *Validated* column reports the number of validating executions that were UNSERIALIZABLE. Across all experiments, *all but three predicted executions were successfully validated* as UNSERIALIZABLE.

The *Diverged* column shows that, in many cases, the validating execution diverged, i.e., it could not match the predicted execution history (§5). Unsurprisingly, the relaxed boundary experienced significantly more divergence than the strict boundary. However, divergence rarely resulted in failed validation: Among the 81 divergent executions across Tables 4 and 5, only three failed validation (i.e., produced SERIALIZABLE executions). One validation failure was caused by divergent behavior

[5]More specifically, the initial state transaction t_0 and the writing transaction t_w constitute the only pair of conflicting writes. If a transaction t_r reads from the initial state, then a commit order with t_r preceding t_w is acyclic. On the other hand, if t_r reads from another transaction, a commit order t_r following t_w is acyclic.

Table 4. IsoPredict effectiveness and performance **under CAUSAL**. "T/O" means the solver did not finish within 24 hours. "Unk" means the solver returned "unknown" without reaching the timeout.

	Prediction	**Prediction**			**Validation**		**Constraint gen.**		**Solving time**	
Program	**strategy**	Unk	Unsat	Sat	Validated	(Diverged)	# Literals	Time	Sat	Unsat
Smallbank	Exact-Strict	0	6	4	**4**	(0)	140 K	8.8 s	13.9 s	11.3 s
	Approx-Strict	0	6	4	**4**	(1)	366 K	22.9 s	1.0 s	3.2 s
	Approx-Relaxed	0	0	10	**9**	(1)	366 K	22.9 s	0.6 s	–
Voter	Exact-Strict	0	10	0	**0**	(0)	687 K	61.7 s	–	64.5 s
	Approx-Strict	0	10	0	**0**	(0)	1,526 K	131.7 s	–	10.4 s
	Approx-Relaxed	0	10	0	**0**	(0)	1,526 K	132.1 s	–	10.0 s
TPC-C	Exact-Strict	0	1	9	**9**	(0)	3,493 K	220.4 s	230.4 s	752.3 s
	Approx-Strict	0	1	9	**9**	(0)	6,508 K	425.8 s	35.1 s	105.2 s
	Approx-Relaxed	0	0	10	**10**	(0)	6,508 K	425.5 s	22.7 s	–
Wikipedia	Exact-Strict	1	9	0	**0**	(0)	180 K	13.9 s	–	24.0 s
	Approx-Strict	0	10	0	**0**	(0)	529 K	36.3 s	–	1.3 s
	Approx-Relaxed	0	8	2	**2**	(1)	529 K	36.3 s	2.5 s	1.0 s

(a) Small workload

	Prediction	**Prediction**			**Validation**		**Constraint gen.**		**Solving time**	
Program	**strategy**	T/O	Unsat	Sat	Validated	(Diverged)	# Literals	Time	Sat	Unsat
Smallbank	Exact-Strict	4	1	5	**5**	(1)	1,073 K	55.6 s	8,618.9 s	2,366.2 s
	Approx-Strict	1	0	9	**9**	(0)	2,175 K	121.0 s	332.5 s	–
	Approx-Relaxed	0	0	10	**10**	(0)	1,073 K	118.8 s	19.3 s	–
Voter	Exact-Strict	9	1	0	**0**	(0)	2,623 K	235.1 s	–	5,708.7 s
	Approx-Strict	0	10	0	**0**	(0)	5,623 K	490.5 s	–	47.2 s
	Approx-Relaxed	0	10	0	**0**	(0)	5,623 K	496.1 s	–	47.1 s
TPC-C	Exact-Strict	4	3	3	**3**	(0)	36,434 K	1,914.6 s	30,413.1 s	24,281.2 s
	Approx-Strict	2	0	8	**8**	(0)	60,834 K	3,416.1 s	1,210.3 s	–
	Approx-Relaxed	0	0	10	**10**	(2)	60,834 K	3,332.3 s	186.2 s	–
Wikipedia	Exact-Strict	8	1	1	**1**	(0)	1,773 K	111.9 s	910.2 s	1,876.8 s
	Approx-Strict	0	9	1	**1**	(0)	4,316 K	263.7 s	15.6 s	30.1 s
	Approx-Relaxed	0	8	2	**2**	(2)	4,316 K	258.3 s	20.3 s	25.3 s

(b) Large workload

unrelated to aborts (§5), and the other two failures were caused by previously aborted transactions being committed (an implementation issue discussed in §6).

Performance. The four rightmost columns of each table report the performance of IsoPredict's predictive analysis, which consists of two components: (1) the time the analysis takes to generate SMT constraints (*Constraint gen.*) and (2) SMT solving time (*Solving time*). Each table also reports the size of the generated constraints (*# Literals*),[6] which correlates with constraint generation time. SMT solving is significantly faster for successful prediction (*Sat*) than for failed prediction (*Unsat*),[7] so the table reports the two average solving times separately.

Compared to the other prediction strategies, Exact-Strict, which generates a single quantified constraint, spends less time generating constraints but more time solving constraints because its constraints are inherently harder to solve. Approx-Relaxed and Approx-Strict have performance similar to each other, which makes sense since they share the same approximation techniques.

[6]The Approx-Strict and Approx-Relaxed prediction strategies generate different constraints, but they have the same size.
[7]It makes sense that successful prediction, which finds a single satisfying solution, is faster than failed prediction, which requires the solver to prove that no satisfying solution exists.

Table 5. IsoPredict effectiveness and performance **under RC**. "T/O" means the solver did not finish within 24 hours. "Unk" means the solver returned "unknown" without reaching the timeout.

	Prediction	Prediction			Validation		Constraint gen.		Solving time	
Program	**strategy**	Unk	Unsat	Sat	Validated	(Diverged)	# Literals	Time	Sat	Unsat
Smallbank	Exact-Strict	0	0	10	**10**	(0)	144 K	10.0 s	2.3 s	–
	Approx-Strict	0	0	10	**10**	(0)	370 K	24.3 s	0.8 s	–
	Approx-Relaxed	0	0	10	**10**	(0)	370 K	24.4 s	0.6 s	–
Voter	Exact-Strict	0	0	10	**10**	(2)	688 K	62.5 s	12.9 s	–
	Approx-Strict	0	0	10	**10**	(7)	1,527 K	133.0 s	12.3 s	–
	Approx-Relaxed	0	0	10	**10**	(10)	1,527 K	132.7 s	12.7 s	–
TPC-C	Exact-Strict	0	0	10	**10**	(0)	3,855 K	359.0 s	52.0 s	–
	Approx-Strict	0	0	10	**10**	(0)	6,869 K	569.2 s	27.2 s	–
	Approx-Relaxed	0	0	10	**10**	(3)	6,869 K	588.9 s	22.8 s	–
Wikipedia	Exact-Strict	2	1	7	**7**	(2)	184 K	15.4 s	3.9 s	8.0 s
	Approx-Strict	0	3	7	**7**	(1)	533 K	38.3 s	2.1 s	0.5 s
	Approx-Relaxed	0	3	7	**7**	(7)	533 K	38.1 s	1.7 s	0.5 s

(a) Small workload

	Prediction	Prediction			Validation		Constraint gen.		Solving time	
Program	**strategy**	T/O	Unsat	Sat	Validated	(Diverged)	# Literals	Time	Sat	Unsat
Smallbank	Exact-Strict	0	0	10	**9**	(1)	1,085 K	60.3 s	624.6 s	–
	Approx-Strict	0	0	10	**10**	(1)	2,187 K	124.5 s	19.0 s	–
	Approx-Relaxed	0	0	10	**10**	(1)	2,187 K	128.7 s	51.7 s	–
Voter	Exact-Strict	0	0	10	**10**	(2)	2,625 K	255.7 s	212.0 s	–
	Approx-Strict	0	0	10	**10**	(6)	5,625 K	491.7 s	76.2 s	–
	Approx-Relaxed	0	0	10	**10**	(10)	5,625 K	495.2 s	75.4 s	–
TPC-C	Exact-Strict	0	0	10	**10**	(2)	38,062 K	2,571.0 s	898.6 s	–
	Approx-Strict	0	0	10	**10**	(2)	62,462 K	3,981.9 s	279.7 s	–
	Approx-Relaxed	0	0	10	**10**	(4)	62,462 K	4,040.4 s	201.0 s	–
Wikipedia	Exact-Strict	0	0	10	**10**	(1)	1,807 K	124.6 s	81.4 s	–
	Approx-Strict	0	0	10	**10**	(1)	4,350 K	272.8 s	29.2 s	–
	Approx-Relaxed	0	0	10	**9**	(10)	4,350 K	272.9 s	16.9 s	–

(b) Large workload

Generating constraints can take a long time—often longer than constraint-solving time. We investigated this issue by using the *perf* [41] and *py-spy* [21] performance analysis tools on the slowest instance of constraint generation: the large workload of TPC-C under RC using the Approx-Relaxed strategy (Table 5). To the best of our understanding, 97% of time is spent in Python code (IsoPredict and Z3Py), and 3% is spent in C code (Z3). Of the time spent in Python, 81% is spent in Z3Py functions, with most time spent in the following Z3PY API functions and their callees: __call__(), And(), and Or(). The __call__() function is part of Z3Py's implementation of SMT functions, which act as callable objects in Python. The And() and Or() functions create conjunction and disjunction clauses, respectively. Z3Py functions call into Z3 code written in C; an unknown fraction of the time spent in Z3Py is due to making cross-language calls from Z3Py to Z3.

7.3 Comparison with MonkeyDB

MonkeyDB is a transactional key–value data store that aims to produce unusual executions that are legal under a target isolation level [7]. MonkeyDB handles each read to a key by returning a randomly chosen value among the set of legal values under the target isolation level.

MonkeyDB and IsoPredict both aim to find erroneous executions under weak isolation, but they use completely different approaches. MonkeyDB relies on a customized query engine that produces

Table 6. Comparison between MonkeyDB [7] and IsoPredict (Approx-Relaxed strategy) **under CAUSAL**. The numbers report how often a benchmark assertion failed (*Fail*) or the history was UNSERIALIZABLE (*Unser*).

	MonkeyDB		**IsoPredict**
Program	Fail	Unser	Unser
Smallbank	70%	98%	90%
Voter	70%	80%	0%
TPC-C	98%	100%	100%
Wikipedia	0%	11%	20%

(a) Small workload

	MonkeyDB		**IsoPredict**
Program	Fail	Unser	Unser
Smallbank	84%	100%	100%
Voter	56%	80%	0%
TPC-C	100%	100%	100%
Wikipedia	0%	19%	20%

(b) Large workload

Table 7. Comparison between MonkeyDB [7], IsoPredict (Approx-Strict strategy), and regular execution using MySQL **under RC**. Each number is the percentage of runs in which a benchmark assertion failed (*Fail*) or the history was UNSERIALIZABLE (*Unser*).

	MonkeyDB		**IsoPredict**	**MySQL**
Program	Fail	Unser	Unser	Fail
Smallbank	100%	100%	100%	0%
Voter	89%	100%	100%	0%
TPC-C	100%	100%	100%	50%
Wikipedia	54%	54%	70%	0%

(a) Small workload

	MonkeyDB		**IsoPredict**	**MySQL**
Program	Fail	Unser	Unser	Fail
Smallbank	100%	100%	100%	0%
Voter	95%	100%	100%	0%
TPC-C	100%	100%	100%	70%
Wikipedia	89%	89%	100%	0%

(b) Large workload

a single execution, while IsoPredict uses predictive analysis to analyze an equivalence class of many executions at once. They also differ in how they define and expose UNSERIALIZABLE behavior: IsoPredict tries to find an UNSERIALIZABLE execution, while MonkeyDB uses programmer-crafted assertions to detect UNSERIALIZABLE behaviors.

Tables 6 and 7 compare MonkeyDB and IsoPredict's effectiveness at predicting UNSERIALIZABLE executions. To account for MonkeyDB's randomized approach, we ran it 100 times for each configuration: 10 times for each of the 10 RNG seeds used as benchmark input (§7.1). The percentage of these executions with an assertion failure is reported in the *Fail* column.

To compare MonkeyDB and IsoPredict directly, we computed whether each execution produced by MonkeyDB was UNSERIALIZABLE, by generating and solving constraints corresponding to the definition of SERIALIZABLE. An assertion failure is a sufficient but unnecessary condition for an UNSERIALIZABLE execution; hence, for MonkeyDB, the number of executions failing assertions (*Fail*) never exceeds the number of UNSERIALIZABLE executions (*Unser*).

The *IsoPredict* column shows the percentage of executions that led to UNSERIALIZABLE predictions that were successfully validated (i.e., same results as the *Validation* columns in Tables 4 and 5). The tables use the best-performing prediction strategy for each isolation level.

Quantitatively, MonkeyDB and IsoPredict are comparable, finding erroneous executions at similar rates, except for two cases. In one case—Voter under CAUSAL—MonkeyDB produces UNSERIALIZABLE executions, but IsoPredict never predicts any. Voter issues only one write transaction under SERIALIZABLE (Algorithm 3), from which it is impossible to predict an UNSERIALIZABLE execution under CAUSAL, because IsoPredict cannot predict events that did not happen in the observed execution. In contrast, since MonkeyDB chooses values on the fly, its choices of reads can lead Voter to perform additional writes, leading to UNSERIALIZABLE behavior. In another case—Wikipedia under CAUSAL—IsoPredict is able to predict several UNSERIALIZABLE executions while MonkeyDB never has assertion failures, since its assertions are not sensitive enough to detect UNSERIALIZABLE behaviors.

Qualitatively, the approaches differ in two significant ways. First, IsoPredict does not require programmers to write assertions. Second and more significantly, IsoPredict predicts UNSERIALIZABLE executions from observed executions, which in theory could be produced by any data store. In contrast, MonkeyDB's approach requires its specialized query engine.

Comparison with regular execution. Both MonkeyDB and IsoPredict routinely produce UNSERIALIZABLE executions for the evaluated programs, but a natural question is whether executing these programs normally on a real-world data store yields UNSERIALIZABLE executions. To evaluate this question, we executed the programs using MySQL [37] in RC mode (MySQL does not support CAUSAL). As for the MonkeyDB runs, we executed each program 100 times—10 times for each of the 10 RNG seeds used as input to the program—and evaluated the assertions used by MonkeyDB.

Table 7's *MySQL* columns show the percentage of runs in which an assertion failed, a sufficient condition for an UNSERIALIZABLE history. The results show that Smallbank, Voter, and Wikipedia never experienced an assertion failure under regular execution.[8] TPC-C experienced an assertion failure half of the time on the small workload and 70% of the time on the large workload. In contrast, MonkeyDB and IsoPredict often produce assertion-failing, UNSERIALIZABLE executions.

Differences between our MonkeyDB results and the MonkeyDB paper's results. In our experiments, MonkeyDB triggered fewer assertion failures than reported in the MonkeyDB paper [7]. These differences exist because we found and fixed a few bugs in the ported benchmarks and their assertions, which eliminated a few spurious failures. We confirmed all of the bugs and fixes with the MonkeyDB authors [8]. To be clear, the differences do not impact the MonkeyDB paper's takeaway: MonkeyDB often produces UNSERIALIZABLE, erroneous executions for the evaluated programs.

8 RELATED WORK

The closest existing approaches to IsoPredict are arguably MonkeyDB [7], IsoDiff [23], 2AD [51], and Sinha et al.'s predictive analysis [46]. As §7.3 explained, MonkeyDB produces a single execution, which may or may not be unserializable, while IsoPredict predicts unserializable executions from an observed execution. IsoPredict can in theory work with any data store that can generate execution traces, while MonkeyDB requires its specialized query engine.

IsoDiff and 2AD detect unserializable behaviors based on an observed execution [23, 51]. They build an abstract graph that does not take into account potential dependencies between read values. As the 2AD paper acknowledges, "2AD's abstract histories are value-agnostic and do not account for control flow within a program; in effect, 2AD's abstract history construction process assumes that each variable read and written can assume arbitrary values. However, there are often dependencies (e.g., $y = x + 1$) between the values that variables assume" [51]. As a result, 2AD incurs high false positive rates even after using programmer-guided refinement: 37 reported "witnesses" on average per application, but only 22 bugs across 12 applications, or 2 bugs on average per application [51].

In contrast, IsoPredict accounts for dependencies among read values through its axiomatic encoding of constraints, which permits encoding of potential dependencies using the prediction boundary. IsoPredict may still report false positives, but for narrower reasons: divergent aborts or (only when using the relaxed boundary) intra-transaction dependencies.

Sinha et al.'s analysis predicts atomicity violations in shared-memory multithreaded programs by encoding the conditions for unserializability as SMT constraints [46]. A key difference with IsoPredict is that Sinha et al.'s work deals with execution histories of shared-memory programs, in which all pairs of conflicting accesses are fully ordered, while IsoPredict deals with execution

[8]It is an open question whether MySQL in RC mode can actually produce UNSERIALIZABLE executions for these programs. Data store implementations may preclude behaviors that are theoretically possible under the target isolation level.

histories of distributed data store applications, in which conflicting accesses are *unordered* in general. As a result, Sinha et al.'s work only needs to encode graph cyclicity, while IsoPredict must encode that *every* potential commit order is acyclic. Addressing this unique challenge led us to develop IsoPredict's approximate encoding (§4.2.2). Other differences include the different prediction spaces: Sinha et al.'s analysis predicts different orderings of critical sections on the same lock, while IsoPredict predicts different write–read orders.

Dynamic analysis. Non-predictive dynamic analysis can check if an observed execution satisfies an isolation level. ECRacer checks whether an observed execution is serializable, using a relaxed definition of serializability that accounts for commutative operations [11]. In contrast, IsoPredict finds *new* executions that violate serializability.

Prior work uses run-time testing and constraint solving to check if a data store provides a stated weak isolation level [6, 32, 48, 52, 53]. In contrast, IsoPredict assumes the data store provides the target weak isolation level and predicts feasible unserializable executions.

Model checking explores multiple executions, avoiding exhaustively exploring all possible executions by using techniques such as dynamic partial order reduction (DPOR) [1, 10, 27]. Conschecker uses a DPOR-based stateless model checking algorithm to verify distributed shared-memory programs under causal consistency [1]. Bouajjani et al.'s work adapts DPOR-based algorithms to transactional database applications to check them under a range of isolation levels [10].

Static analysis. Static analysis can find unserializable behavior, but precision and performance scale poorly with program size. C^4 and Nagar and Jagannathan's analysis detect serializability violations under causal consistency, eventual consistency, and snapshot isolation [12, 39]. Clotho uses static analysis, model checking, and test generation to detect unserializable executions; it avoids false positives by verifying the feasibility of unserializable behaviors [43]. In contrast, IsoPredict detects unserializable behaviors with high precision by basing it on a single observed execution.

Isolation levels. IsoPredict generates constraints based on isolation levels encoded in Biswas and Enea's axiomatic framework [6]. Other prior work besides Biswas and Enea's has introduced axiomatic encodings of weak isolation levels [9, 14, 31, 42].

Adya et al. define various isolation levels with dependency graphs where each level allows certain types of cycles [2]. Their approach encompasses "classical" database isolation levels such as read committed and snapshot isolation, but not isolation levels typically used in distributed data stores such as causal consistency [4, 9, 13, 29, 42] and eventual consistency [13].

IsoPredict currently supports only CAUSAL and RC, by encoding axioms from Biswas and Enea's framework [6]. We expect that extending IsoPredict to more isolation levels from their framework—READ ATOMIC (a.k.a. REPEATED READS) and SNAPSHOT ISOLATION—to be straightforward. We do not know how difficult it would be to encode other isolation levels (e.g., EVENTUAL CONSISTENCY and MONOTONIC ATOMIC VIEW) into Biswas and Enea's framework or into IsoPredict.

9 CONCLUSION

IsoPredict is the first predictive analysis for detecting unserializable behaviors of applications backed by weakly isolated data stores. IsoPredict's design introduces novel approaches to address challenges involving constraint complexity, constraint encoding, and divergent behaviors. An evaluation shows that, based on observed executions of data store applications, IsoPredict effectively, precisely, and efficiently predicts feasible, unserializable behaviors.

DATA-AVAILABILITY STATEMENT

An artifact reproducing this paper's results is publicly available [24].

ACKNOWLEDGMENTS

We thank the MonkeyDB authors [7] for making their implementation publicly available and answering our questions about it; Vincent Beardsley and Noah Charlton for helpful discussions; and the anonymous reviewers for valuable feedback. This material is based in part upon work supported by the National Science Foundation under Grant Numbers NSF CCF-2118745, CSR-2106117, and OAC-2112606, and by Oracle America, Inc.

REFERENCES

[1] Parosh Abdulla, Mohamed Faouzi Atig, S. Krishna, Ashutosh Gupta, and Omkar Tuppe. 2023. Optimal Stateless Model Checking for Causal Consistency. In *Tools and Algorithms for the Construction and Analysis of Systems*, Sriram Sankaranarayanan and Natasha Sharygina (Eds.). Springer Nature Switzerland, Cham, 105–125.

[2] A. Adya, B. Liskov, and P. O'Neil. 2000. Generalized isolation level definitions. In *Proceedings of 16th International Conference on Data Engineering (Cat. No.00CB37073)*. IEEE Computer Society, Los Alamitos, CA, USA, 67–78. https://doi.org/10.1109/ICDE.2000.839388

[3] Mustaque Ahamad, Gil Neiger, James E. Burns, Prince Kohli, and P.W. Hutto. 1995. Causal Memory: Definitions, Implementation and Programming. *Distributed Computing* 9, 1 (1995), 37–49. https://doi.org/10.1007/BF01784241

[4] Jade Alglave, Luc Maranget, and Michael Tautschnig. 2014. Herding Cats: Modelling, Simulation, Testing, and Data Mining for Weak Memory. *ACM Trans. Program. Lang. Syst.* 36, 2, Article 7 (Jul 2014), 74 pages. https://doi.org/10.1145/2627752

[5] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. 1995. A Critique of ANSI SQL Isolation Levels. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data* (San Jose, California, USA) *(SIGMOD '95)*. ACM, New York, NY, USA, 1–10. https://doi.org/10.1145/223784.223785

[6] Ranadeep Biswas and Constantin Enea. 2019. On the Complexity of Checking Transactional Consistency. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 165 (Oct 2019), 28 pages. https://doi.org/10.1145/3360591

[7] Ranadeep Biswas, Diptanshu Kakwani, Jyothi Vedurada, Constantin Enea, and Akash Lal. 2021. MonkeyDB: Effectively Testing Correctness under Weak Isolation Levels. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 132 (Oct 2021), 27 pages. https://doi.org/10.1145/3485546

[8] Ranadeep Biswas, Diptanshu Kakwani, Jyothi Vedurada, Constantin Enea, and Akash Lal. 2023. Personal communication.

[9] Ahmed Bouajjani, Constantin Enea, Rachid Guerraoui, and Jad Hamza. 2017. On verifying causal consistency. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages* (Paris, France) *(POPL '17)*. Association for Computing Machinery, New York, NY, USA, 626–638. https://doi.org/10.1145/3009837.3009888

[10] Ahmed Bouajjani, Constantin Enea, and Enrique Román-Calvo. 2023. Dynamic Partial Order Reduction for Checking Correctness against Transaction Isolation Levels. *Proc. ACM Program. Lang.* 7, PLDI, Article 129 (Jun 2023), 26 pages. https://doi.org/10.1145/3591243

[11] Lucas Brutschy, Dimitar Dimitrov, Peter Müller, and Martin Vechev. 2017. Serializability for Eventual Consistency: Criterion, Analysis, and Applications. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages* (Paris, France) *(POPL '17)*. Association for Computing Machinery, New York, NY, USA, 458–472. https://doi.org/10.1145/3009837.3009895

[12] Lucas Brutschy, Dimitar Dimitrov, Peter Müller, and Martin Vechev. 2018. Static Serializability Analysis for Causal Consistency. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Philadelphia, PA, USA) *(PLDI 2018)*. Association for Computing Machinery, New York, NY, USA, 90–104. https://doi.org/10.1145/3192366.3192415

[13] Sebastian Burckhardt. 2014. Principles of Eventual Consistency. *Found. Trends Program. Lang.* 1, 1–2 (oct 2014), 1–150. https://doi.org/10.1561/2500000011

[14] Andrea Cerone, Giovanni Bernardi, and Alexey Gotsman. 2015. A Framework for Transactional Consistency Models with Atomic Visibility. In *26th International Conference on Concurrency Theory (CONCUR 2015) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 42)*, Luca Aceto and David de Frutos Escrig (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 58–71. https://doi.org/10.4230/LIPIcs.CONCUR.2015.58

[15] Chaoyi Cheng, Mingzhe Han, Nuo Xu, Spyros Blanas, Michael D. Bond, and Yang Wang. 2023. Developer's Responsibility or Database's Responsibility? Rethinking Concurrency Control in Databases. In *13th Conference on Innovative Data Systems Research, CIDR 2023, Amsterdam, The Netherlands, January 8-11, 2023*. www.cidrdb.org. https://www.cidrdb.org/cidr2023/papers/p30-cheng.pdf

[16] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, JJ Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li,

Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. 2012. Spanner: Google's Globally-Distributed Database. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*. USENIX Association, Hollywood, CA, 261–264. https://www.usenix.org/conference/osdi12/technical-sessions/presentation/corbett

[17] Natacha Crooks, Youer Pu, Lorenzo Alvisi, and Allen Clement. 2017. Seeing is Believing: A Client-Centric Specification of Database Isolation. In *Proceedings of the ACM Symposium on Principles of Distributed Computing* (Washington, DC, USA) *(PODC '17)*. ACM, New York, NY, USA, 73–82. https://doi.org/10.1145/3087801.3087802

[18] Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and Jakob Rehof (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 337–340.

[19] Djellel Eddine Difallah, Andrew Pavlo, Carlo Curino, and Philippe Cudre-Mauroux. 2013. OLTP-Bench: An Extensible Testbed for Benchmarking Relational Databases. *Proc. VLDB Endow.* 7, 4 (Dec 2013), 277–288. https://doi.org/10.14778/2732240.2732246

[20] Mostafa Elhemali, Niall Gallagher, Nick Gordon, Joseph Idziorek, Richard Krog, Colin Lazier, Erben Mo, Akhilesh Mritunjai, Somasundaram Perianayagam, Tim Rath, Swami Sivasubramanian, James Christopher Sorenson III, Sroaj Sosothikul, Doug Terry, and Akshat Vig. 2022. Amazon DynamoDB: A Scalable, Predictably Performant, and Fully Managed NoSQL Database Service. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. USENIX Association, Carlsbad, CA, 1037–1048. https://www.usenix.org/conference/atc22/presentation/elhemali

[21] Ben Frederickson. 2024. https://github.com/benfred/py-spy

[22] Leonidas Galanis, Supiti Buranawatanachoke, Romain Colle, Benoît Dageville, Karl Dias, Jonathan Klein, Stratos Papadomanolakis, Leng Leng Tan, Venkateshwaran Venkataramani, Yujun Wang, and Graham Wood. 2008. Oracle Database Replay. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data* (Vancouver, Canada) *(SIGMOD '08)*. Association for Computing Machinery, New York, NY, USA, 1159–1170. https://doi.org/10.1145/1376616.1376732

[23] Yifan Gan, Xueyuan Ren, Drew Ripberger, Spyros Blanas, and Yang Wang. 2020. IsoDiff: Debugging Anomalies Caused by Weak Isolation. *Proc. VLDB Endow.* 13, 12 (Jul 2020), 2773–2786. https://doi.org/10.14778/3407790.3407860

[24] Chujun Geng, Spyros Blanas, Michael D. Bond, and Yang Wang. 2024. *IsoPredict artifact.* https://doi.org/10.5281/zenodo.10802748

[25] Chujun Geng, Spyros Blanas, Michael D. Bond, and Yang Wang. 2024. IsoPredict: Dynamic Predictive Analysis for Detecting Unserializable Behaviors in Weakly Isolated Data Store Applications. arXiv:2404.04621 Extended version of PLDI 2024 paper.

[26] Chujun Geng, Spyros Blanas, Michael D. Bond, and Yang Wang. 2024. *IsoPredict implementation.* https://github.com/PLaSSticity/IsoPredict-implementation

[27] M. Ghafoor, M. Mahmood, and J. Siddiqui. 2016. Effective Partial Order Reduction in Model Checking Database Applications. In *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE Computer Society, Los Alamitos, CA, USA, 146–156. https://doi.org/10.1109/ICST.2016.25

[28] Seth Gilbert and Nancy Lynch. 2002. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News* 33 (June 2002), 51–59. Issue 2. https://doi.org/10.1145/564585.564601

[29] Jad Hamza. 2015. *Algorithmic Verification of Concurrent and Distributed Data Structures.* Ph. D. Dissertation. PhD thesis, Université Paris Diderot.

[30] Jeff Huang, Patrick O'Neil Meredith, and Grigore Rosu. 2014. Maximal sound predictive race detection with control flow abstraction. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Edinburgh, United Kingdom) *(PLDI '14)*. Association for Computing Machinery, New York, NY, USA, 337–348. https://doi.org/10.1145/2594291.2594315

[31] Gowtham Kaki, Kapil Earanky, KC Sivaramakrishnan, and Suresh Jagannathan. 2018. Safe replication through bounded concurrency verification. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 164 (Oct 2018), 27 pages. https://doi.org/10.1145/3276534

[32] Kyle Kingsbury and Peter Alvaro. 2020. Elle: Inferring Isolation Anomalies from Experimental Observations. *Proc. VLDB Endow.* 14, 3 (Nov 2020), 268–280. https://doi.org/10.14778/3430915.3430918

[33] Dileep Kini, Umang Mathur, and Mahesh Viswanathan. 2017. Dynamic race prediction in linear time. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Barcelona, Spain) *(PLDI 2017)*. Association for Computing Machinery, New York, NY, USA, 157–170. https://doi.org/10.1145/3062341.3062374

[34] K. R. M. Leino and Clément Pit-Claudel. 2016. Trigger Selection Strategies to Stabilize Program Verifiers. In *Computer Aided Verification*, Swarat Chaudhuri and Azadeh Farzan (Eds.). Springer International Publishing, Cham, 361–381.

[35] Qian Li, Peter Kraft, Michael Cafarella, Çağatay Demiralp, Goetz Graefe, Christos Kozyrakis, Michael Stonebraker, Lalith Suresh, Xiangyao Yu, and Matei Zaharia. 2023. R3: Record-Replay-Retroaction for Database-Backed Applications. *Proc. VLDB Endow.* 16, 11 (Jul 2023), 3085–3097. https://doi.org/10.14778/3611479.3611510

[36] P. Mahajan, L. Alvisi, and M. Dahlin. 2011. *Consistency, Availability, Convergence.* Technical Report TR-11-22. Computer Science Department, University of Texas at Austin.

[37] MySQL 2023. http://www.mysql.com

[38] MySQL 2023. MySQL Cluster. https://www.mysql.com/products/cluster/

[39] Kartik Nagar and Suresh Jagannathan. 2018. Automated Detection of Serializability Violations under Weak Consistency. arXiv:1806.08416 [cs.PL]

[40] Andrew Pavlo. 2017. What Are We Doing With Our Lives? Nobody Cares About Our Concurrency Control Research. In *Proceedings of the 2017 ACM International Conference on Management of Data* (Chicago, Illinois, USA) *(SIGMOD '17)*. Association for Computing Machinery, New York, NY, USA, 3. https://doi.org/10.1145/3035918.3056096

[41] perf 2024. https://perf.wiki.kernel.org/index.php/Main_Page

[42] Matthieu Perrin, Achour Mostefaoui, and Claude Jard. 2016. Causal Consistency: Beyond Memory. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Barcelona, Spain) *(PPoPP '16)*. Association for Computing Machinery, New York, NY, USA, Article 26, 12 pages. https://doi.org/10.1145/2851141.2851170

[43] Kia Rahmani, Kartik Nagar, Benjamin Delaware, and Suresh Jagannathan. 2019. CLOTHO: Directed Test Generation for Weakly Consistent Database Systems. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 117 (Oct 2019), 28 pages. https://doi.org/10.1145/3360543

[44] Jake Roemer, Kaan Genç, and Michael D. Bond. 2020. SmartTrack: efficient predictive race detection. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 747–762. https://doi.org/10.1145/3385412.3385993

[45] Mahmoud Said, Chao Wang, Zijiang Yang, and Karem Sakallah. 2011. Generating Data Race Witnesses by an SMT-Based Analysis. In *NASA Formal Methods*, Mihaela Bobaru, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 313–327. https://doi.org/10.1007/978-3-642-20398-5_23

[46] Arnab Sinha, Sharad Malik, Chao Wang, and Aarti Gupta. 2012. Predicting Serializability Violations: SMT-Based Search vs. DPOR-Based Search. In *Hardware and Software: Verification and Testing*, Kerstin Eder, João Lourenço, and Onn Shehory (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 95–114. https://doi.org/10.1007/978-3-642-34188-5_11

[47] Snowflake 2023. Snowflake transactions. https://docs.snowflake.com/en/sql-reference/transactions

[48] Cheng Tan, Changgeng Zhao, Shuai Mu, and Michael Walfish. 2020. COBRA: making transactional key-value stores verifiably serializable. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation (OSDI'20)*. USENIX Association, USA, Article 4, 18 pages. https://www.usenix.org/conference/osdi20/presentation/tan

[49] Chuzhe Tang, Zhaoguo Wang, Xiaodong Zhang, Qianmian Yu, Binyu Zang, Haibing Guan, and Haibo Chen. 2022. Ad Hoc Transactions in Web Applications: The Good, the Bad, and the Ugly. In *Proceedings of the 2022 International Conference on Management of Data* (Philadelphia, PA, USA) *(SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 4–18. https://doi.org/10.1145/3514221.3526120

[50] Hünkar Can Tunç, Umang Mathur, Andreas Pavlogiannis, and Mahesh Viswanathan. 2023. Sound Dynamic Deadlock Prediction in Linear Time. *Proc. ACM Program. Lang.* 7, PLDI, Article 177 (Jun 2023), 26 pages. https://doi.org/10.1145/3591291

[51] Todd Warszawski and Peter Bailis. 2017. ACIDRain: Concurrency-Related Attacks on Database-Backed Web Applications. In *Proceedings of the 2017 ACM International Conference on Management of Data* (Chicago, Illinois, USA) *(SIGMOD '17)*. ACM, New York, NY, USA, 5–20. https://doi.org/10.1145/3035918.3064037

[52] Rachid Zennou, Ranadeep Biswas, Ahmed Bouajjani, Constantin Enea, and Mohammed Erradi. 2022. Checking Causal Consistency of Distributed Databases. *Computing* 104, 10 (Oct 2022), 2181–2201. https://doi.org/10.1007/s00607-021-00911-3

[53] Jian Zhang, Ye Ji, Shuai Mu, and Cheng Tan. 2023. Viper: A Fast Snapshot Isolation Checker. In *Proceedings of the Eighteenth European Conference on Computer Systems* (Rome, Italy) *(EuroSys '23)*. Association for Computing Machinery, New York, NY, USA, 654–671. https://doi.org/10.1145/3552326.3567492

Received 2023-11-03; accepted 2024-03-31