ETH zürich

ETH Library

# Effective Serializability for Eventual Consistency

**Report**

**Author(s):**
Brutschy, Lucas; Dimitrov, Dimitar; Müller, Peter; Vechev, Martin

**Publication date:**
2016

**Permanent link:**
https://doi.org/10.3929/ethz-a-010634886

**Rights / license:**
In Copyright - Non-Commercial Use Permitted

This page was generated automatically upon download from the ETH Zurich Research Collection.
For more information, please consult the Terms of use.

# Effective Serializability for Eventual Consistency

Lucas Brutschy
Department of Computer Science, ETH Zürich
lucas.brutschy@inf.ethz.ch

Dimitar Dimitrov
Department of Computer Science, ETH Zürich
dimitar.dimitrov@inf.ethz.ch

Peter Müller
Department of Computer Science, ETH Zürich
peter.mueller@inf.ethz.ch

Martin Vechev
Department of Computer Science, ETH Zürich
martin.vechev@inf.ethz.ch

## ABSTRACT

Developing and reasoning about systems using eventually consistent data stores is a difficult challenge because these systems can sometimes exhibit weakly consistent behaviors that are unexpected and difficult to understand.

This paper makes several contributions which advance our conceptual understanding and reasoning capabilities of eventually consistent systems: (i) a new serializability criterion which generalizes conflict serializability, but is based on a dependency model with two algebraic properties: commutativity and absorption; our model enables precise reasoning about programs that use high-level replicated data types, common in modern systems; (ii) two dynamic analysis algorithms for detecting violations of our criterion; (iii) a complete implementation of the analysis algorithm.

We performed a thorough experimental evaluation on two realistic use cases: debugging cloud-backed mobile applications and implementing clients of a popular eventually consistent key-value store. Our experimental results indicate that our criterion matches the programmers' notion of correctness, is more effective in finding bugs than prior approaches, and can be used during the development of practical applications under weak guarantees.

## 1. INTRODUCTION

Modern distributed systems increasingly rely on replicated data stores [7, 13, 26, 14] in order to achieve high scalability and availability. As dictated by the CAP theorem [11], consistency, availability and partition-tolerance cannot be achieved at the same time. While various trade-offs exist, most replicated stores tend to provide relaxed correctness notions that are variants of eventual consistency: updates are not immediately but eventually propagated to other replicas, and replicas observing the same set of operations reflect the same state.

However, relaxations of strong consistency come at a price as applications may now experience unexpected behaviors that are not possible under strong consistency. These behaviors may lead to serious errors, and make the development of such applications more challenging.

In such cases, it is tempting to require that applications guarantee by themselves a stronger notion of consistency, like serializability. One can then reason about application correctness without considering the effects of weak consistency, and importantly serializability violations can guide the derivation of correct synchronization where needed. Such violations, however, are very difficult to detect in general (in fact, NP-hard [21]). This is where stronger serializability criteria, like conflict serializability, come into play.

*Challenges.* To be practically useful, a serializability criterion must possess at least three properties. First, it must be general in the sense that it supports reasoning about a wide range of practical data stores and operations. Second, it must be *weak enough* so it rules out few desirable behaviors. And third, it must be *strong enough* so that it can be checked efficiently on realistic systems. Previous approaches are typically restricted to systems with guarantees stronger than eventual consistency, can only reason about primitive reads and writes, or are computationally difficult to check.

*This work.* We propose a new serializability criterion for eventually consistent systems that satisfies the above three properties. Technically, our criterion generalizes the classic notion of conflict serializability [21]: (i) to deal with relaxed behaviors induced by eventually consistent data stores, while (ii) handling high-level replicated data types (such as replicated maps and lists [23, 5]), which are commonly used in modern distributed applications. Our generalization leverages commutativity of operations, and the fact that some operations mask the effects of others. This allows us to permit executions not possible under strong consistency, yet equivalent to executions that are strongly consistent. Since our criterion only assumes eventual consistency, it immediately applies to all consistency levels that strengthen eventual consistency in various ways [27, 16, 5, 6].

To substantiate the usefulness of our criterion, we built a dynamic analyzer that checks whether the criterion holds on program executions, and evaluate our analyzer on two application domains. First, we analyzed 33 mobile apps written in TouchDevelop [28], a framework which uses weakly

consistent cloud types [5]. The experimental results indicate that our serializability criterion captures the programmers' intentions. Moreover, our analyzer found violations of the criterion leading to errors in the applications. Second, we implemented the database benchmark TPC-C [29] using the eventually consistent data store RIAK and show how our criterion can guide developers to derive correct client implementations.

We note that our serializability criterion need not be used on the entire application (as this is likely to be too restrictive). Instead, it is most useful when applied to specific parts of the program intended to be serializable (e.g., payment check-out). This usage scenario is in line with how standard conflict serializability is used for shared memory concurrent programming (e.g., [30]).

*Contributions.* The main contributions of our paper are:

- An effective serializability criterion for clients of eventually consistent data stores. Our criterion generalizes conflict serializability to deal with weakly consistent behaviors and high-level data types.
- Polynomial-time algorithms to check whether the criterion holds on a given program execution.
- An implementation of our algorithms for to data stores: the TOUCHDEVELOP cloud platform for mobile device applications, and the distributed database RIAK.
- A detailed evaluation that indicates that our criterion: captures an intuitive understanding of correctness; is useful for finding previously undetected errors; can help in building correct and scalable applications running on eventually consistent data stores.

## 2. OVERVIEW

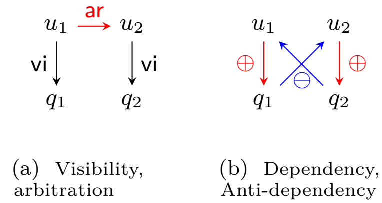
In this section we provide an intuitive understanding of the challenges our approach addresses. Full formal details are provided in later sections.

*Motivating example.* Consider the following code fragment, adapted from a mobile gaming library [1]

```
1  Players.at(G, I).user.setIfEmpty(userID)
2  if Players.at(G, I).user != userID
3    // try next position
```

Here, `Players` is a distributed map, mapping a game `G` and a position `I` to a user participating in the game. A user identifier is stored in the field `user`. Suppose that each operation runs in its own transaction. The intended behavior is that each spot can only be taken by up to one user. Indeed, this is exactly what happens under strong consistency: if two competing accesses to the same spot are performed concurrently, one of the `setIfEmpty` operations will remain without effect, and the corresponding client has to try the next spot in the game.

Suppose we now execute the code using an eventually consistent data store where updates such as `setIfEmpty` are asynchronously executed at other replicas, while queries are only read from the local replica. Figure 1a shows a




(a) Visibility, arbitration (b) Dependency, Anti-dependency

Figure 1: Execution of motivating example, with operations:
$u_1$ `map(G,I).user.setIfEmpty("Destroyer")`
$u_2$ `map(G,I).user.setIfEmpty("Widowmaker")`
$q_1$ `map(G,I).user` returns `"Destroyer"`
$q_2$ `map(G,I).user` returns `"Widowmaker"`

possible execution of such a system. Two users 'Destroyer' and 'Widowmaker' try to acquire the same position `I` in game `G`. The graph denotes with vi whether an operation is observed by a second operation, and by ar the order in which update conflicts are eventually resolved by the system. Here, each query only observes one of the updates, causing both clients to think they acquired the spot in the game. The behavior is not serializable: There is no sequential execution of $u_1, q_1, u_2, q_2$, in which the queries return these values.

In this work we use the classical notions of dependency and anti-dependency to characterize serializability: intuitively, a query depends on an update which is visible, if the result of the query would change had the update become invisible. Similarly, a query anti-depends on an update that is invisible, if the query result would change had the update become visible. Figure 1b shows the dependencies $\oplus$ and anti-dependencies $\ominus$ in our example. If the four relations of dependency, anti-dependency, program order po and arbitration order ar form a cycle, then the execution is *not serializable.*

*Key challenge.* A precise serializability criterion requires a precise notion of dependency. For reads and writes, this is fairly straightforward: a read depends on the last-arbitrated write that is visible to it and accesses the same record. For high-level operations such as counters, maps, sets and tables, however, this is not as straightforward. Consider for instance the case where there is a second function in the above gaming library which lists all games a player participates in:

```
1  Games := Players.select(_.user == userID)
```

Suppose now, user 'Destroyer' reserves a spot in a game via the update `setIfEmpty("Destroyer")`, and at the same time 'Widowmaker' lists the games that he participates in via the above query. Here, Destroyer's update is not a dependency of Widowmaker's query as no matter whether it is visible or not the result of the query will be the same: Destroyer's reservation does not influence Widowmaker's participation. The reson is that, even though both operations access the same data, they actually commute ($uq \equiv qu$). Thus, determining dependencies precisely requires reasoning about commutativity.In this work we show how to leverage commutativity properties of arbitrary operations in order to capture the dependencies between them.

While useful, commutativity alone is not sufficient. Consider again the execution shown in Figure 1b, but suppose that now, we had used a `set` instead of a `setIfEmpty`. Even though both updates do not commute with both queries, the

[1]"cloud game lobby", written in TOUCHDEVELOP and available on `http://touchdevelop.com`.

execution is serializable in the order $u_1q_1u_2q_2$. This is because $u_2$ hides the effect of $u_1$ to $q_2$, and therefore $q_2$ does not depend on $u_1$. We call this absorption, $u_1u_2 \equiv u_2$. Again, if the operations are reads and writes, absorption is easy to define: updates absorb each other if and only if they are non-commutative. However, for the richer operations considered in this work (e.g., those of high level data types), the definition of absorption may be more involved: arbitrary operations such as `setIfEmpty` may be non-absorbing, partially absorbing, or absorbing only under specific conditions.

*Summary.* In summary, this paper introduces a new serializability criterion for programs using eventual consistency based on both, commutativity and absorption, enabling precise reasoning of arbitrary operations in both transactional and non-transactional programs. In what follows, we formally present our model, state our main theorem and evaluate the approach on several realistic data stores.

*Related work.* Checking for serializability is NP-hard [21] in general. Conflict serializability defines a stronger criterion on executions in an attempt to be computationally feasible (also [21]) by using a conflict relation between operations which was first defined via basic reads and writes, but also can be defined through commutativity [31]. However, it assumes a serial schedule where all conflicts are resolved, and is not therefore not applicable to executions in weakly consistent systems.

Several works have provided serializability conditions on executions on data stores with various guarantees, e.g., Snapshot Isolation [9] as well as a variety of weak memory models (e.g., [25, 20, 2]). As with our criterion, these are typically based on detecting cycles in graphs involving some notion of dependency and anti-dependency. The main differences to our work are that (a) they assume stronger consistency guarantees provided by the data store, and (b) they use low-level read and write reasoning instead of algebraic reasoning. Our work is a generalization of these previous criteria that makes them applicable to a broader class of real-world systems.

Similarly to our work, Fekete et al [9] use serializability checking to check an implementation of the TPC-C database benchmark. However, they use static checking and a database guaranteeing snapshot isolation, while our work is based on dynamic checking and an eventually consistent database. Zellag et al. [32] use a criterion similar to [9] to quantify the anomalies in applications using eventually consistent data stores. They do not prove the criterion correct w.r.t. eventual consistency and reason only about reads and writes.

Several works suggest reasoning about the preservation of integrity invariants in weakly consistent data stores (see, e.g., [1]). While reasoning about integrity invariants directly can allow more behaviors than serializability, and can also lead to additional performance gains due to weak replication, it requires detailed manual specifications which are notoriously difficult to provide. However, light annotations of which parts of the application should be serializable are practically useful and easy to provide. We thus provide this option to developers, a capability similar to how atomicity annotations were used in [30] for shared memory concurrency.

# 3. WEAKLY CONSISTENT SYSTEMS

We continue with a model of weakly consistent systems, which we will later use to reason about serializability. Our model is loosely based on [4]. We consider a system of processes that interact with a weakly consistent data store. Interaction between a process and the store happens in a sequence of atomic actions issued to manipulate the stored data. Our interest lies in the possible behaviors of such a system, which we model as a set of the action histories.

## 3.1 Actions and traces

An action represents an atomic operation performed by a process against the replicated data store. For example, one could set a given record $\mathsf{x}$ to zero, or also, observe that the record holds the value zero. These would be the $\mathsf{x.set}(0)$ and the $\mathsf{x.get}():0$ actions. More formally, an *action* is an operation combined with concrete argument and return values.

We assume that action semantics is given as a prefix-closed set of *legal action sequences*. For example, $\mathsf{x.set}(0)\ \mathsf{x.get}():0$ would be legal, while $\mathsf{x.set}(0)\ \mathsf{x.get}():1$ would be not. Under this style of specification, two action sequences $\alpha$ and $\beta$ are *equivalent* iff they are legal in exactly the same contexts:

$$\alpha \equiv \beta \text{ iff } \{(\chi, \gamma) \mid \chi\alpha\gamma \text{ legal}\} = \{(\chi, \gamma) \mid \chi\beta\gamma \text{ legal}\}.$$

For example, two actions might commute, or one action might (right-)absorb another, as expressed by the equivalences:

$$\mathsf{x.set}(0)\ \mathsf{y.set}(1) \equiv \mathsf{y.set}(1)\ \mathsf{x.set}(0)$$
$$\mathsf{z.set}(0)\ \mathsf{z.set}(1) \equiv \mathsf{z.set}(1)$$

### 3.1.1 Traces

The order of commuting actions in a given sequence is irrelevant, and we will prefer to work with traces [19] instead of sequences. A trace relaxes the total order that actions have in a sequence to a partial one, such that all non-commuting actions nevertheless remain ordered. With the switch to a partial order, we will refer to action occurrences as *events*. The partial order itself has to be *lower-finite*, i.e., every event has to be preceded by finitely many others.

*Definition 1.* A *trace* is a lower-finite partial order $\tau$ of a countable set $E$ of events such that for all $f, g \in E$:

(1.1) $f \xrightarrow{\tau} g$ or $fg \equiv gf$ or $g \xrightarrow{\tau} f$, and

(1.2) if $f \xrightarrow{\tau} g$ but $f \xrightarrow{\tau} h \xrightarrow{\tau} g$ for no $h \in E$, then $fg \not\equiv gf$.

The first condition ensures that the trace orders all pairs of non-commuting actions, while the second one ensures that no unnecessary ordering is introduced. Below is an example of a trace where the record $\mathsf{x}$ gets incremented twice:

$$\begin{array}{ccc} & \mathsf{x.get}():0 & \\ \swarrow & & \searrow \\ \mathsf{x.add}(1) & & \mathsf{x.add}(2) \\ \searrow & & \swarrow \\ & \mathsf{x.get}():3 & \end{array}$$

Similarly to taking subsequences, we can restrict a trace to a subset of its events. Then, (1.1) will still hold but (1.2) does not need to. For example, if we restrict the above trace to the two $\mathsf{get}$ actions, then they remain ordered even though they commute. Such restrictions will be useful later, and we will call them *semi-traces*.

Operations on sequences other than restriction transfer to semi-traces too. For example, the above trace equals

the concatenation of its prefix x.get():0 → x.add(1) and its suffix x.add(2) → x.get():3. We can thus speak of semi-trace legality, and also of semi-trace equivalence.

### 3.1.2 Updates and queries

In order to simplify our arguments, we assume that actions divide into either updates or queries. An *update* may modify the store but does not indicate a return value. On the other hand, a *query* may not modify the store but may indicate a return value. Furthermore, we assume that an update can always be applied, i.e., that it is free of any pre-conditions. Our assumptions are non-restrictive as any action can be split into an query after an update, and also, any update can be made to skip if its pre-condition is not met.

## 3.2 Histories and schedules

As standard, we model a *process* in the system as a possibly infinite sequence of action events. Thus, each process has a definitive start but need not have an end. A *transaction* is a contiguous segment of a specific process, and is intended to execute atomically with respect to the other system processes. Taken together, these components form a history:

*Definition 2.* A *history* $(E, \mathsf{po}, T)$ consists of

– a countable set $E$ of events (each labeled by an action),

– a partial ordering po that partitions $E$ into processes,

– a partition $T$ of the processes into transactions.

The externally observable behavior of the system is characterized by the set of histories that the system can possibly leave. To prevent undesired behaviors, the store may put some constraints on this set. For example, it could allow only histories where transactions are atomic. As usual, we attribute such guarantees to whether a history has a schedule of a specific kind. More specifically, we are interested in two kinds of schedules: serial ones and eventually consistent ones.

*Definition 3.* A *serial schedule* of a history $(E, \mathsf{po}, T)$ is a linear ordering so of $E$ such that:

(3.1) the union $\mathsf{po} \cup \mathsf{so}$ is lower-finite and acyclic,

(3.2) every prefix of so is legal, and

(3.3) no two transactions $t_1 \neq t_2 \in T$ overlap, i.e., either:

(a) $f \xrightarrow{\mathsf{so}} g$ for all $f \in t_1$, $g \in t_2$, or

(b) $g \xrightarrow{\mathsf{so}} f$ for all $g \in t_2$, $f \in t_1$.

A history is *serializable* if it has a serial schedule, that is, if its transactions are atomic in a linear order. Serializability eases reasoning about concurrent processes dramatically, but is typically too expensive to be ensured by a system in a replicated setting. That is why many data stores ensure the much weaker but cheaper property of eventual consistency. Data store clients then have to deal with atomicity issues by some other means.

We consider what is sometimes called strong eventual consistency [24]. Informally, it is a combination of two properties: first, every process observes a consistent view but only of a subset of the updates in the system so far; second, every update eventually propagates to the view of every process.

| Relation | Type | Description |
|---|---|---|
| po | $E \times E$ | process ordering |
| so | $E \times E$ | serial ordering |
| vi | $U \times Q$ | update visibility |
| ar | $U \times U$ | update arbitration |
| $\oplus$ | $U \times Q$ | dependency |
| $\ominus$ | $Q \times U$ | anti-dependency |

Table 1: Model relations over a set $E = U \cup Q$ of update $U$ and query $Q$ events.

*Definition 4.* An *eventually consistent schedule* $(\mathsf{vi}, \mathsf{ar})$ of a history $(E, \mathsf{po}, T)$ consists of

– a relation vi indicating the updates that queries observe,

– trace ar of the updates in $E$ arbitrating their order,

and such that they meet three conditions:

(4.1) the union $\mathsf{po} \cup \mathsf{vi}$ is lower-finite and acyclic,

(4.2) each query $q$ is legal in ar restricted to $\{u \mid u \xrightarrow{\mathsf{vi}} q\}$,

(4.3) for any update $u$ the set $\{q \mid u \not\xrightarrow{\mathsf{vi}} q\}$ is finite.

The first two conditions are analogous to the ones for serial schedules, and deal with consistency. The third condition deals with update propagation. For brevity, we will omit "eventually consistent" and just use the term *schedule*. An example of a schedule is given in Figure 1.

Note that any serial schedule has an implicit visibility relation and an implicit arbitration trace. Moreover, for them the conditions (4.1)-(4.3) hold automatically, thus the serial schedule can be seen as an eventually consistent one.

Whenever (4.1) holds but (4.3) or (4.2) possibly do not, we will speak of a *pre-schedule*. If it happens that (4.2) holds for a specific query $q$, then we will say that $q$ is *legal* in the pre-schedule. These technicalities prove to be quite useful in the next section, where we reason about serializability.

# 4. A SERIALIZABILITY CRITERION

We will present a sufficient criterion that tells whether a history is serializable given one of its schedules. The main idea is to derive a certain digraph, known as the dependency serialization graph, from the schedule. The digraph relates all transactions in the history so that their lower-finite topological orderings correspond to serial schedules. Thus, acyclicity becomes a sufficient condition for serializability.

## 4.1 Dependency

The main motive in our criterion is the notion of a query depending on an update. A query in a schedule depends on an update if this update is visible and it may influence the query legality. For example, consider the serial schedule

x.add(1) y.set(1) x.set(0) x.add(2) x.get():2 y.get():1.

Here, x.get():2 depends on both x.set(0) and x.add(2). If we remove any of them, the the query becomes illegal. However, if we remove any other actions, then the query remains legal. Similarly, y.get():1 depends only on y.set(1).

There could be different reasons why an update happens to be a dependency, or rather, why it fails to be one. In order

to decide that, we will use two properties: commutativity and absorption. Recall that given two actions $f$ and $g$

1. $f$ and $g$ commute iff $fg \equiv gf$;

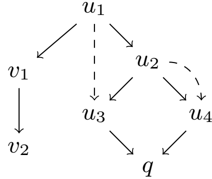
2. $f$ is absorbed by $g$ iff $fg \equiv g$.

In example above, $\mathsf{y.set}(0)$ is not a dependency of $\mathsf{x.get}()\text{:}2$ as it commutes with all updates on $\mathsf{x}$. Thus, the schedule is equivalent to one where $\mathsf{y.set}(0)$ is not visible to $\mathsf{x.get}()\text{:}2$

$$\mathsf{x.add}(1)\ \mathsf{x.set}(0)\ \mathsf{x.add}(2)\ \mathsf{x.get}()\text{:}2\ \mathsf{y.set}(1)\ \mathsf{y.get}()\text{:}1.$$

Here, $\mathsf{x.set}(0)$ absorbs $\mathsf{x.add}(1)$. Thus, $\mathsf{x.add}(1)$ is not a dependency of $\mathsf{x.get}()\text{:}2$ as no matter whether it is present or not we get an equivelent serial schedule.

Now, let us consider an eventually consistent schedule $(\mathsf{vi}, \mathsf{ar})$. To find out the dependencies of a query $q$ we restrict $\mathsf{ar}$ to the updates that $q$ observes. We then append $q$ and inspect the result. Consider, e.g.,



Here, solid arrows indicate the trace order, and dashed arrows indicate absorption. The two updates $v_1$ and $v_2$ are not dependencies of $q$ because they do not precede it in the trace order, and therefore can be "moved past" $q$. The update $u_2$ is not a dependency of $q$ either as it gets absorbed by the adjacent update $u_4$. But after that $u_1$ and $u_3$ become adjacent, and so $u_1$ can get absorbed by $u_3$. What remains are the two dependencies $u_3$ and $u_4$ of $q$.

We capture the above observations with two operations and that remove non-dependencies from a given visibility relation: one for commutativity, and one for absorption. With their help, we define the dependency relation of a pre-schedule:

*Definition 5.* For a given pre-schedule $(\mathsf{vi}, \mathsf{ar})$ we define two operations over relations $R$ from updates to queries:

(5.1) $(u, q) \in R^\sharp$ iff $u \xrightarrow{R} q$ and there is an update $v \xrightarrow{R} q$ such that $vq \not\equiv qv$, and either $u \xrightarrow{\mathsf{ar}} v$ or $u = v$.

(5.2) $(u, q) \in R'$ iff $u \xrightarrow{R} q$ and also for all updates $v \xrightarrow{\mathsf{vi}^\sharp} q$ if $u \xrightarrow{\mathsf{ar}} v$ but $u \xrightarrow{\mathsf{ar}} w \xrightarrow{\mathsf{ar}} v$ for no $w \xrightarrow{R} q$, then $uv \not\equiv v$.

The *dependency relation* of a pre-schedule $(\mathsf{vi}, \mathsf{ar})$ is the largest $\oplus \subseteq \mathsf{vi}^\sharp$ such that $\oplus = \oplus'$.

The main property of the dependency relation is that making any set of non-dependencies invisible to a query preserves the query legality:

Theorem 1. *Given some pre-schedule* $(\mathsf{vi}, \mathsf{ar})$*, for every relation* $R$ *such that* $\oplus \subseteq R \subseteq \mathsf{vi}$*, a query* $q$ *is legal in* $(\mathsf{vi}, \mathsf{ar})$ *iff it is legal in the pre-schedule* $(R, \mathsf{ar})$.

Proof. To make our argument simpler, let us introduce some notation. We are looking at relations in the inteval

$$[\oplus, \mathsf{vi}] = \{R \mid \oplus \subseteq R \subseteq \mathsf{vi}\}.$$

For brevity, let us collect all the restrictions of $\mathsf{ar}$ that such a relation $R$ determines into a single map $\langle R \rangle$:

$$q \mapsto \mathsf{ar} \restriction \{u \mid u \xrightarrow{R} q\}.$$

Operations on semi-traces extend pointwisely to such maps, and if we denote $q \mapsto q$ with $Q$, then our claim becomes:

$$\forall R \in [\oplus, \mathsf{vi}].\, \langle R \rangle \cdot Q \text{ is legal} \iff \langle \oplus \rangle \cdot Q \text{ is legal.}$$

The basis of our proof is that $\oplus$ can be derived from any relation $R \in [\oplus, \mathsf{vi}]$ as the limit of the decreasing sequence:

$$R = R_0 \supseteq R_0 \cap \mathsf{vi}^\sharp = R_1 \supseteq R_1' = R_2 \supseteq R_2' = R_3 \supseteq \dots$$

This is a simple consequence of $\mathsf{vi}$ being lower-finite plus a standard fixed-point argument applied to the $'$ operation (see, e.g., [10, Proposition II-2.4]). From here we will show that every step $i$ above preserves legality in the sense that

$$\langle R \rangle \cdot Q \text{ is legal} \iff \langle R_i \rangle \cdot Q \text{ is legal.}$$

That turns to be sufficient as the above property is preserved when taking limits, again a consequence of lower-finiteness.

So let us make the first step. The $\mathsf{vi}^\sharp$ relation splits $\langle R \rangle$ into two parts, the second one commuting with $Q$:

$$\langle R \rangle \cdot Q \equiv \langle R \cap \mathsf{vi}^\sharp \rangle \cdot \langle R \setminus \mathsf{vi}^\sharp \rangle \cdot Q \equiv \langle R \cap \mathsf{vi}^\sharp \rangle \cdot Q \cdot \langle R \setminus \mathsf{vi}^\sharp \rangle.$$

Since updates are free of pre-conditions, the right-most side is legal iff its prefix $\langle R \cap \mathsf{vi}^\sharp \rangle \cdot Q$ is legal.

Now, the remaining steps. It is enough to show that the restrictions of $\mathsf{ar}$ that they give rise to are all equivalent. Let us first study how updates get absorbed when applying the $'$ operation. Recall definition (5.2), and consider the relation:

$$u \prec_i^q v \text{ iff } (u, q) \in R_i \setminus R_{i+1} \text{ breaks (5.2) because of } v \xrightarrow{\mathsf{vi}^\sharp} q.$$

Because absorption is transitive, this relation is transitive in the sense that for all $i > j$:

$$u \prec_i^q v \prec_j^q w \implies u \prec_i^q w.$$

This allows us to conclude that if we remove $(u, q)$ at step $i$, then it is always because of some $v$ visible to $q$ at that step:

$$(u, q) \in R_i \setminus R_{i+1} \implies u \prec_i^q v \xrightarrow{R_i} q \text{ for some } v.$$

Indeed, recursively tracing the reason for removals, we get a sequence of steps $i = i_n \geq i_{n-1} \geq \dots \geq i_1$ and updates:

$$u \prec_{i_n}^q v_n \prec_{i_{n-1}}^q \cdots \prec_{i_1}^q v_1 = v,$$

such that $v \xrightarrow{R_i} q$ (for otherwise, $v$ was removed even earlier and we can continue the sequence).

We are now ready to prove that $\langle R_i \rangle \equiv \langle R_{i+1} \rangle$ for $i \geq 1$. By our previous argument, the updates removed from their relation $R_i$ with $q$ are the non-maximal vertices of the dag

$$(V = \{u \mid u \xrightarrow{R_i} q\}, E = \prec_i^q).$$

After sorting the non-maximal vertices $u_1, \dots, u_{n(q)}$ in $V$ topologically, consider the sequence of trace restrictions:

$$\alpha_0 = \mathsf{ar} \restriction V \supset \alpha_1 = \alpha_0 \setminus u_1 \supset \dots \supset \alpha_n = \alpha_{n-1} \setminus u_{n(q)}.$$

By definition (5.2), every $\alpha_j$ is of the form $\beta_j u_j v_j \gamma_j$, where $\beta_j v_j \gamma_j = \alpha_{j+1}$ and $u_j \prec_i^q v_j$. Here, $v_j$ absorbs $u_j$, and

therefore $\alpha_j \equiv \alpha_{j+1}$. We conclude that $\alpha_1 \equiv \alpha_{n(q)}$ for every query $q$, or in other words $\langle R_i \rangle \equiv \langle R_{i+1} \rangle$. $\square$

## 4.2 Anti-dependency

The notion of dependency has a natural counterpart called anti-dependency. An anti-dependency of a query in a given schedule is an update not visible to the query but such that making it visible would turn it into a dependency.

*Definition 6.* Given an update and a query $u \overset{\mathsf{vi}}{\not\to} q$ in a pre-schedule $(\mathsf{vi}, \mathsf{ar})$, let $\oplus^{uq}$ be the dependency relation with respect to the modified visibility $\mathsf{vi}^{uq} = \mathsf{vi} \cup (u, q)$. We define the *anti-dependency relation* of the pre-schedule as

$$q \xrightarrow{\ominus} u \text{ iff } u \xrightarrow{\oplus^{uq}} q.$$

The main property of the anti-dependency relation is that making any set of non-anti-dependencies visible to a query introduces no new dependencies:

THEOREM 2. *If* $(\mathsf{vi}_1, \mathsf{ar})$ *and* $(\mathsf{vi}_2, \mathsf{ar})$ *are two pre-schedules such that* $\mathsf{vi}_1 \subseteq \mathsf{vi}_2$*, and* $\ominus_1^{-1} \cap \mathsf{vi}_2 = \varnothing$*, then* $\oplus_1 \supseteq \oplus_2$*.*

PROOF. Reasoning about commutativity is more or less straightforward, and we will focus on absorbtion only, i.e., assume $\mathsf{vi}_1 = \mathsf{vi}_1^\sharp$, and $\mathsf{vi}_2 = \mathsf{vi}_2^\sharp$.

Consider any two pre-schedules $(\mathsf{vi}_X, \mathsf{ar})$ and $(\mathsf{vi}_Y, \mathsf{ar})$, such that $\mathsf{vi}_X \subseteq \mathsf{vi}_Y$. The respective operations $'^X$ and $'^Y$ possess a kind of monotonicity property. For every pair of relations $A \in [\oplus_X, \mathsf{vi}_X]$ and $B \in [\oplus_Y, \mathsf{vi}_Y]$, update $u$, and a query $q$

$$A(u,q) \supseteq B(u,q) \implies A'^X(u,q) \supseteq B'^Y(u,q),$$

where $R(u,q)$ stands for the set $\{v \xrightarrow{R} q \mid u \xrightarrow{\mathsf{ar}} v \text{ or } u = v\}$. By the fixed-point argument from the proof of Theorem 1, this property transfers to the respective fixed-points:

$$A(u,q) \supseteq B(u,q) \implies \oplus_X(u,q) \supseteq \oplus_Y(u,q).$$

Moving on to the two pre-schedules $(\mathsf{vi}_1, \mathsf{ar})$ and $(\mathsf{vi}_2, \mathsf{ar})$, suppose that $(u,q) \in \mathsf{vi}_2$. As $\mathsf{vi}_1(v,p) \supseteq \oplus_1^{uq}(v,p)$ for every pair $(v,p) \in \mathsf{vi}_1$, we conclude that

$$\oplus_1(v,p) \supseteq \oplus_1^{uq}(v,p).$$

But $q \overset{\ominus_1}{\not\to} u$, i.e., $u \overset{\oplus_1^{uv}}{\not\to} q$, and therefore $\oplus_1 \supseteq \oplus_1^{uq}$. With this fact at hand, we will prove that for all $(u,q) \in \oplus_2$

$$\oplus_1(u,q) \supseteq \oplus_2(u,q).$$

Proceeding by well-founded induction on the set $\oplus_2(u,q)$, let $v \neq u$ belong to it. By the inductive hypothesis:

$$\oplus_1(u,v) \supseteq \oplus_1(v,q) \supseteq \oplus_2(v,q) \ni v.$$

We can, therefore, conclude that $\oplus_1(u,q) \setminus u \supseteq \oplus_2(u,q) \setminus u$. Now, if we let $R = \oplus_1 \cup (u,q)$, then we obtain:

$$R(u,q) \supseteq \oplus_2(u,q)$$

Because the relation $R$ belongs to the interval $[\mathsf{vi}_1^{uq}, \oplus_1^{uq}]$, the monotonicity property applies here, and so:

$$\oplus_1(u,q) \supseteq \oplus_1^{uq}(u,q) \supseteq \oplus_2(u,q).$$

$\square$

## 4.3 Serializability

Our serializability criterion essentially provides a sufficient condition for when a given schedule can be converted into a serial one. We require that the serial schedule preserves four relations: $\mathsf{po}$, $\mathsf{ar}$, $\oplus$ and $\ominus$. The first one, $\mathsf{po}$, has to be preserved by any serial schedule. Preserving the other relations is just a choice that automatically implies that any serial pre-schedule $\mathsf{so} \supseteq \mathsf{po} \cup \mathsf{ar} \cup \oplus \cup \ominus$ is in fact a schedule. To check whether such a pre-schedule exists we consider the digraph obtained from the above union by collapsing every transaction into a vertex:

*Definition 7.* The *dependency serialization graph* of a given pre-schedule $(\mathsf{vi}, \mathsf{ar})$ of a history $(E, \mathsf{po}, T)$ is a digraph over the set of transaction $T$, where an arc $(s,t)$ is present iff the union $\mathsf{po} \cup \mathsf{ar} \cup \oplus \cup \ominus$ relates an event $f \in s$ to another $g \in t$.

It turns out that the required serial schedule $\mathsf{so}$ exists if the dependency serialization graph contains no cycles. This leads to the following serializability crierion:

THEOREM 3. *A history* $(E, \mathsf{po}, T)$ *with a finite number of processes is serializable if it has a schedule* $(\mathsf{vi}, \mathsf{ar})$ *with an acyclic dependency serialization graph.*

PROOF. Suppose that there exists a serial pre-schedule $\mathsf{so} \supseteq \mathsf{po} \cup \mathsf{ar} \cup \oplus \cup \ominus$. Let us denote its visibility relation with $\mathsf{vi}_{\mathsf{so}}$. The intersection $\mathsf{vi}_\cap = \mathsf{vi} \cap \mathsf{vi}_{\mathsf{so}}$ meets the condition $\oplus \subseteq \mathsf{vi}_\cap \subseteq \mathsf{vi}$ of Theorem 1, and therefore every query in $E$ is legal in the pre-schedule $(\mathsf{vi}_\cap, \mathsf{ar})$. In turn, this pre-schedule meets the condition of Theorem 2 with respect to $(\mathsf{vi}_{\mathsf{so}}, \mathsf{ar})$, and therefore $\oplus_{\mathsf{so}} \subseteq \oplus_\cap \subseteq \mathsf{vi}_{\mathsf{so}}$. Applying Theorem 1 once more, we conclude that $\mathsf{so}$ is a schedule.

We still need to establish that such a pre-schedule $\mathsf{so}$ exists. Because the dependency serialization graph is acyclic and there is a finite number of processes, we could use a simple round-robin scheduler to produce $\mathsf{so}$. We only need to prove that every transaction $t$ has a finite number of predecessors.

Because the schedule $(\mathsf{vi}, \mathsf{ar})$ is eventually consistent, all the relations $\mathsf{po}$, $\mathsf{ar}$, $\oplus$, and $\ominus$ are lower-finite. This implies that every transaction has only a finite number of immediate predecessors. Thus, if a transaction $s$ precedes $t$, then the process of $s$ has a $\mathsf{po}$-last transaction that does the same (for otherwise we would run into a cycle). Because $\mathsf{po}$ is lower-finite, and also, there are finitely may processes, we conclude that only finitely many transactions precede $t$. $\square$

# 5. DETECTION ALGORITHM

In this section, we give two algorithms for detecting serializability violations in a schedule $(\mathsf{vi}, \mathsf{ar})$. The first one is general, while the second makes assumptions on the data types but is asymptotically more efficient. Detecting serializability violations amounts to determining the dependencies $\oplus$ and anti-dependencies $\ominus$ of the pre-schedule, and performing cycle detection. The latter has well-known linear-time solutions, we focus on computing $\oplus$ and $\ominus$ here.

Our algorithms assume, for each data type, two specifications, *commutativity specification* $\sharp$ and the *absorption specification* $\rhd$, which give sufficient conditions for commutativity and absorption:

$$u \sharp v \implies uv \equiv vu,$$
$$u \rhd v \implies uv \equiv v.$$

In practice, for each pair of operations, we provide a first-order logic formula that can can be checked in constant time

| | $\mathsf{put}_x[k', v']$ | $\mathsf{get}_x[k', v']$ | $\mathsf{size}_x[n']$ |
|---|---|---|---|
| $\mathsf{put}_x[k, v]$ | $k \neq k'$ or $v = v'$ | $k \neq k'$ | never |
| $\mathsf{get}_x[k, v]$ | $k \neq k'$ | always | always |
| $\mathsf{size}_x[n]$ | never | always | always |

Figure 2: Commutativity of dictionary actions.

**Algorithm 1** Generic algorithm for determining the dependencies for a schedule with updates $U$, queries $Q$, visibility $\mathsf{vi}$ and arbitration $\mathsf{ar}$

1: **function** DEPENDENCIES($U, Q, \mathsf{vi}, \mathsf{ar}$)
2: $(\oplus, \ominus) \leftarrow (\emptyset, \emptyset)$
3: **for** $q \in Q$ **do**
4: $U_r \leftarrow \{u \in U \mid (u, q) \in \mathsf{vi}^\sharp\}$
5: $(U_p, E_p) \leftarrow$ PRUNE $(U_r, \mathsf{ar} \upharpoonright U_r)$
6: $\oplus \leftarrow \oplus \cup (U_p \times \{q\})$
7: **for** $u \in \{u \in U \mid u \overset{\mathsf{vi}}{\not\to} q\}$ **do**
8: $(U_u, E_u) \leftarrow$ PRUNE $(U_p \cup u, \mathsf{ar} \upharpoonright U_p \cup u)$
9: **if** $u \in U_u$ **then**
10: $\ominus \leftarrow \ominus \cup (q, u)$
11: **return** $(\oplus, \ominus)$

12: **function** PRUNE($V, E$)
13: $E \leftarrow$ TREDUCTION($E$)
14: $W \leftarrow E$
15: **while** $W \neq \emptyset$ **do**
16: $((u_1, u_2) : W) \leftarrow W$
17: **if** $u_1 \rhd u_2$ **then**
18: $(V, E, N) \leftarrow$ DELFROMRED($V, E, u_1$)
19: $W \leftarrow E \cap (W \cup N)$
20: **return** $(V, E)$

given the arguments and return values of the two operations. For an example of a very simple commutativity specification for a dictionary, see Figure 2.

## 5.1 Generic algorithm

Algorithm 1 gives an algorithm that directly applies the results of Section 4.1. Here, PRUNE (line 12) computes the fixed-point $\oplus = \oplus'$ using a standard work-list algorithm. In every step of the work-list computation, it is checked whether an update is absorbed by its successor in the transitively reduced arbitration order. If so, DELFROMRED in line 18 removes the update $u$ from a transitively reduced graph $(V, E)$, inserts edges from all predecessors to all successors of $u$, recomputes the transitive reduction, and returns all newly inserted edges. It thereby preserves both reachability and transitive reduction over removals.

DEPENDENCIES uses PRUNE to compute both $\oplus$ and $\ominus$. For each query $q$, it first determines the set $U_r$ of all updates that may directly or independently form dependencies of $q$. $U_r$ corresponds to all updates related with $q$ by $\mathsf{vi}^\sharp$ in Definition 5. It then uses PRUNE to eliminate all absorbed updates. All remaining updates are dependencies of $q$. In the second step (line 7 onwards), it reinserts one invisible update after the other, and checks whether it is absorbed by the dependencies of $q$. If not, it is added to the anti-dependencies. The complexity of the algorithm is $O(n^4 m)$ where $n = |U| + |Q|$ and $m = |\mathsf{ar}|$.

**Algorithm 2** Optimized algorithm for determining the dependencies for an execution with updates $U$, queries $Q$, visibility $\mathsf{vi}$, and arbitration $\mathsf{ar}$

1: **function** FASTDEPENDENCIES($U, Q, \mathsf{vi}, \mathsf{ar}$)
2: $\mathsf{ca} \leftarrow$ TCLOSURE($\mathsf{vi}$)
3: $(\oplus, \ominus) \leftarrow (\emptyset, \emptyset)$
4: **for** $q \in Q$ **do**
5: $U_r \leftarrow \{u \in U \mid (u, q) \in \mathsf{vi}^\sharp\}$
6: **for** $u \in (U \setminus U_r)$ **do**
7: **if** $u \overset{\mathsf{vi}}{\not\to} q \wedge u \not\sharp q \wedge q \overset{\mathsf{ca}}{\not\to} u$ **then**
8: $\ominus \leftarrow \ominus \cup (q, u)$
9: **for** $u, u' \in U_r$ with $(u, u') \in \mathsf{ar}$ **do**
10: **if** $u \blacktriangleright u' \wedge u' \overset{\mathsf{vi}}{\to} q$ **then**
11: $U_r \leftarrow U_r \setminus \{u\}$
12: **for** $u \in U_r$ **do**
13: **if** $u \overset{\mathsf{vi}}{\to} q$ **then**
14: $\oplus \leftarrow \oplus \cup (u, q)$
15: **else**
16: $\ominus \leftarrow \ominus \cup (q, u)$
17: **return** $(\oplus, \ominus)$

## 5.2 Optimized algorithm

To derive an asymptotically more efficient algorithm, we will make use of a strengthened ('long-reaching') absorption relation $u \blacktriangleright v \iff \forall \chi \in U^*. u\chi v \equiv \chi v$. Using $\blacktriangleright$ allows us to be agnostic to what happens in between two updates when pruning absorbed operations, relieving us of the task of recomputing the transitive reduction in every step. If we have $\blacktriangleright = \rhd$, the algorithm remains precise. This is true for many practical systems, including the two discussed in the following sections. It is not true, e.g. if a system contains a $\mathsf{swap}$ operation that atomically swaps to fields of a record. Here, $\rhd$ and $\blacktriangleright$ differ: Whether $u$ absorbs $v$ may depend on whether a $\mathsf{swap}$ took place in-between.

The algorithm is listed in Algorithm 2. First, it computes $U_r$, the set of all updates that may form direct or indirect dependencies, or indirect anti-dependencies, as in the generic version. We then check for all other updates, whether they are invisible and non-commutative with $q$, and thus form direct anti-dependencies. For a significant constant speed-up, we filter updates that are guaranteed to happen in the future (after $q$) by the causality $ca$, the transitive closure of $vi$. In the second step (line 9), all updates absorbed by $q$-visible successors in the arbitration order are eliminated. Depending on them being visible to $q$ or not, the remaining elements of the set are added as dependencies and anti-dependencies, resp., in the final step. The complexity of the algorithm is $O(\min(nm, n^3))$, where $n = |U| + |Q|$ and $m = |\mathsf{ar}|$, and thereby significantly faster than the generic version.

# 6. APPLICATION: DEBUGGING CLOUD-BACKED MOBILE SOFTWARE

In this section, we describe and evaluate a dynamic analysis tool for checking serializability issues in TOUCHDEVELOP [28] applications. TOUCHDEVELOP is a platform for mobile device applications providing direct integration of replicated cloud-backed storage. We compare our results to the notion of commutativity races [8] and show that our criterion is better suited for debugging, as it captures harmful violations more

precisely: over all applications, our criterion flags 75% less potential serializability violations.

First, we describe the TouchDevelop system briefly. Then, we discuss a prototype implementation of our tool ECRacer. Finally, we discuss the results of our study of and the serializability violations found.

While we focus on a relatively narrow type applications targeting a specific system, the ideas in this section are generally applicable to a large class of the so called causally consistent data stores [16, 17, 5, 22].

## 6.1 Cloud types

TouchDevelop uses the global sequence protocol [5] to implement a replication system providing *prefix consistency*. In a prefix-consistent system, a client observes a global prefix of all updates including its own ones. This property is stronger than causal consistency, but weaker than snapshot isolation [6]. All three are stronger than eventual consistency and therefore our criterion can be used directly.

All TouchDevelop code executes within weak transactions that provide *atomic visibility*, that is, they guarantee a stable view of a prefix-consistent snapshot of the data store. Updates propagate asynchronously to other clients at the end of each transaction. Transaction boundaries are inserted whenever the runtime is idle, e.g., between execution of event handlers or during execution of blocking operations.

The replication system is exposed to the programmer as *cloud types*: data types that behave similarly to regular heap-stored data structures, but are replicated automatically to other clients. Cloud types include high-level data structures such as maps and lists, but also simple data types with a richer set of atomic operations. For example, a cloud integer can be set to a certain value using set, but also supports a commutative add operation.

To synchronize, clients can query whether their last update on a cloud type is *confirmed*, meaning that the update was included in the global prefix, and all operations that precede it in the prefix are visible to the client.

## 6.2 Prototype implementation

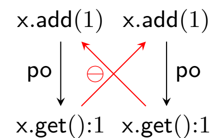
Our tool ECRacer performs dynamic off-line serializability analysis based on the algorithm in Section 5.2. First, the TouchDevelop client runtime is instrumented to record the execution schedule of the a client. Second, a analysis back-end reads the recorded schedules of two or more clients and detects serializability violations as discussed in the previous sections. The violations, which are embodied by cycles in DSGs, are then mapped back to source code locations and reported to the user.

*Recording.* The instrumentation of the TouchDevelop runtime records operations and stores them locally. To reconstruct the visibility relation vi between operations in the system, we use vector clocks [18] implemented with the replicated data store of TouchDevelop itself. That is, we keep a replicated a map from client identifiers to integers, and every update to replicated data is instrumented with a double increment to the clients' logical clock. Queries are versioned with the odd numbers between the even versions of updates. This simplification is sound, since the relative ordering of queries does does not matter and it is important to avoid the overhead of incrementing the replicated counter on every query. The correctness of the computed vector clocks is guaranteed by the fact that the effect of a transaction is made atomically visible to other clients. In total, each record consists of (1) unique identifiers for program, client, and transaction, (2) a local operation counter to reconstruct po, (3) logical time in form of a vector clock, (4) operation name, concrete parameters, and return values, to determine absorption and commutativity between operations, and (5) a unique identifier of the AST node issuing the operation to map operations back to program locations. The arbitration order ar is not recorded, as it is not directly known to the client, it can however be partially inferred from operation return values, and otherwise we assume an arbitration by physical time.

*Analysis.* The analysis back-end implements the algorithm from Section 5.2. In addition to the generic analysis framework, the implementation contains a module specific to TouchDevelop, providing both absorption and commutativity specifications for all operations on cloud types and facilities to map analysis results back to Touch Develop ASTs.

## 6.3 Restricting the scope of the criterion

Requiring serializability for all operations in a program is typically too strong because it often involves harmless serializability violations on data displayed to the user. For example, a violation occurs when a counter is incremented by two clients and displayed by one of them while not observing the increments from the other:




To deal with these harmless violations, we suppose a lightweight specification of what subset of the operations in the program requires serializability. We borrow this idea from the atomic sets of [30]. For the sake of the experiment, where no user-written annotations are available, we exclude from our analysis queries issued within rendering/display code, as they are very frequently executed (every time a page is rerendered), guaranteed to have no other side effects in the program and are almost always harmless in practice. We report on serializability violations on all other operations.

## 6.4 Experimental set-up

We analyzed 33 different applications in total. Of them, 24 were written by regular TouchDevelop users; 6 were written by Microsoft employees to showcase or test the cloud-functionality of TouchDevelop (marked with † in Figure 3). In addition, we analyze 3 cases where we fixed some of the bugs that we found (marked with ‡ in the table). Before analysis, each app is modified to operate on cloud data that is disjoint from that of the original app, as to not obstruct real users with the experiments. In addition, we remove any ability of apps to make use of TouchDevelop's ability to work on user-private data instead of the common public data.

Each application is exercised on two client nodes in parallel via our own random exploration tool for roughly 3 minutes. For 4 games (marked with $*$ in Figure 3) more involved interaction was required, and we executed some of the operations manually. The clients are independently restarted

at random during the execution to achieve realistic overlap in their lifetimes. Both clients are located in Europe, while synchronizing through a data center in the US.

## 6.5 Analysis results

We compare our serializability criterion to *commutativity races*. A commutativity race [8] is a pair of non-commutative operations unrelated by causality. Their absence is a sufficient condition for serializability under causal consistency with atomic visibility. (To see this, observe that (1) under atomic visibility, every cycle in the DSG contains at least one $\ominus$ edge, since $\mathsf{ar}, \mathsf{po}$ are acyclic by definition, and $\oplus$ edges cannot introduce cycles as mutual dependency of transactions is impossible under atomic visibility, and (2) every $\ominus$ edge forms a commutativity race under causal consistency.)

Figure 3 lists the result of our experiment. Columns **Ops.** and **Trans.** denote the number of operations and transactions, resp., executed within the analyzed schedule. Column **Time [s]** contains the time it took to analyze the schedule on a system equipped with an Intel Core i7-4600U CPU with 2.10GHz and 12GB of memory.

We define the number of serializability violations in a program as the number of $\ominus$ edges involved in cycles in the DSG, mapped from events down to program locations. This is a natural metric, as it overapproximates the number of operations whose order must be fixed by synchronization to resolve the violation. Furthermore, it makes the number of commutativity races, column CR in the table, and serializability violations, column SV, comparable: $\text{SV} \leq \text{CR}$.

## 6.6 Discussion

The experiments show the general trend that significantly fewer serializability violations than commutativity races are reported. We detect 75% less serializability violations than commutativity races. In particular, 21 applications contain commutativity races, but only 8 contain serializability violations. This means that the programmer has to inspect significantly fewer program locations when evaluating the serializability of a system, or none at all.

We inspected the violations reported by our analysis manually and found several real bugs, which we describe in the next section. However, some of the reported serializability violations are harmless. In some cases, this is caused by the way TouchDevelop's operations are implemented. For example, the `clear` operation on a cloud-backed map is not an atomic operation, but is implemented as a sequence of (a) first acquiring all keys in the map and (b) removing every element in the map. Some invocations of `clear` are correctly flagged as non-serializable, as it can easily occur that one client observes the original list with the newly inserted element, while another observes a list with just the inserted element, a behavior that is not serializable. While this is certainly unexpected behavior, it is unlikely that such bugs would be fixed by a developer.

## 6.7 Real serializability violations and their fixes

Out of the eight applications that contained serializability violations, we found four bugs that are likely to be fixed by the developer. In this section, we discuss the four serializability violations found in our experiment, their corresponding fixes and how they relate to the alarms reported by the serializability violation detection.

The bugs share the common trait that the developer mistook weak transactional semantics of TouchDevelop for serializable transactions. We propose bug-fixes and argue that establishing their correctness requires precise serializability reasoning.

*Tetris.* One bug appears in in the game "tetris", in which the following program fragment is executed when a new high score is to be saved to the replicated store:

```
1  if (curScore > cloud.highScore)
2    cloud.highScore := curScore
```

Here, a high score of the player's account is stored in a cloud integer. When a game is completed, its score is compared to the local replica of `highScore`. If it is larger, the highScore is overwritten. The update is later propagated to other clients, potentially overwriting *higher* scores achieved on other clients. When the above transaction is executed concurrently at two clients, the execution schedule shows both a commutativity race and a serialization violation and is thereby detected by ECRacer (see Figure 3, id "gcane").

Implementing a fix is not trivial, as there is no atomic max-function on cloud integers provided by TouchDevelop. A fix can instead make use of high-level data structures to store all scores instead of only the first:

```
1  var scoreRec := cloud.scores.add_row
2  scoreRec.val := curScore
3  while (!scoreRev.val.confirmed)
4    sleep(0.2)
5  var highScore = 0
6  foreach (s in cloud.scores)
7    if (s.val > highScore)
8      highScore = s.val
```

The fix adds the newest score to a replicated list, and then waits until the update is appended to the global prefix. Finally, it selects the highest value among all values stored in the list. The synchronization in lines 2-3 is required to not incorrectly determine that the new score is a high score. As seen in Figure 3, the fix still exhibits a commutativity race between the inserts to the list. However, there is no serialization violation, as the program is serializable.

*Events.* The event management app "Events" is non-serializable when it tries to sort its cloud-backed list of events (we omit the code for space reasons here). The algorithm works by (a) removing all elements from the cloud list, (b) sorting them locally and (c) reinserting all elements into the cloud list. If two such transactions are executed in parallel, all elements in the list will be duplicated, since the concurrent removes are merged, while the re-insertions create new unique elements per client. This bug leads to an exponential growth of elements in the worst case scenario. Both commutativity races and serialization violations are effective in detecting the problem. Again, the best fix for the bug is to compute the desired view after querying the data, in this case, to sort a local copy of the cloud-replicated list before accessing it.

*Sky Locale.* The "Sky Locale" quiz game allows a user to overwrite the account of an existing user, because of an incorrect uniqueness check. The serializability violation is correctly detected by ECRacer, however the fixed version still contains a commutativity race. The essential problem is embodied by the following code:

| ID | Name | Category | Ops. | Trans. | CR | SV | $\frac{SV}{CR}$ [%] | Time [s] |
|---|---|---|---|---|---|---|---|---|
| sxjua | Cloud Paper Scissors † | Game | 244 | 96 | 7 | 2 | 29 | 0.066 |
| uvlma | Color Line ∗ | Game | 5 | 4 | 1 | 0 | 0 | 0.003 |
| ycxbc | guess multi-player demo † | Game | 293 | 66 | 3 | 1 | 33 | 0.104 |
| kqfnc | HackER | Game | 115 | 91 | 12 | 6 | 50 | 0.181 |
| ohgxa | keyboard hero ∗ | Game | 2 | 2 | 0 | 0 | - | 0.001 |
| wccqepeb | Online Tic Tac Toe Multiplayer | Game | 565 | 184 | 57 | 17 | 30 | 0.324 |
| uvjba | pentix ∗ | Game | 6 | 6 | 3 | 0 | 0 | 0.002 |
| padg | sky locale ∗ | Game | 347 | 266 | 2 | 1 | 50 | 0.118 |
| – | sky locale ∗ ‡ | Game | 264 | 195 | 0 | 0 | - | 0.047 |
| gcane | tetris ∗ | Game | 8 | 4 | 2 | 1 | 50 | 0.002 |
| – | tetris ∗ ‡ | Game | 14 | 8 | 1 | 0 | 0 | 0.003 |
| fqaba | Chatter box | Social | 131 | 75 | 3 | 0 | 0 | 0.020 |
| etww | Contest Voting † | Social | 57 | 57 | 0 | 0 | - | 0.065 |
| eijba | ec2 demo chat † | Social | 72 | 36 | 0 | 0 | - | 0.009 |
| gbtxe | Hubstar | Social | 263 | 183 | 0 | 0 | - | 0.120 |
| nggfa | instant poll † | Social | 81 | 81 | 0 | 0 | - | 0.019 |
| qnpge | metaverse | Social | 20 | 4 | 3 | 0 | 0 | 0.005 |
| ruef | Relatd | Social | 118 | 65 | 7 | 0 | 0 | 0.014 |
| cvuz | Super Chat | Social | 170 | 58 | 0 | 0 | - | 0.029 |
| wbuei | unique poll | Social | 166 | 143 | 2 | 0 | 0 | 0.071 |
| qzju | cloud card | Tool | 32 | 8 | 0 | 0 | - | 0.006 |
| kzwue | Cloud Example | Tool | 178 | 170 | 2 | 1 | 50 | 0.099 |
| blqz | cloud list † | Tool | 302 | 261 | 2 | 0 | 0 | 0.082 |
| qwidc | Events | Tool | 1458 | 80 | 5 | 2 | 40 | 0.772 |
| – | Events ‡ | Tool | 520 | 65 | 0 | 0 | - | 0.158 |
| nvoha | expense recorder † | Tool | 67 | 60 | 3 | 0 | 0 | 0.007 |
| wkvhc | Expense Splitter | Tool | 25 | 14 | 0 | 0 | - | 0.007 |
| kmac | FieldGPS | Tool | 12 | 12 | 1 | 0 | 0 | 0.005 |
| kjxzcgcv | NuvolaList 2 | Tool | 297 | 223 | 6 | 0 | 0 | 0.340 |
| eddm | Save Passwords | Tool | 345 | 259 | 0 | 0 | - | 0.118 |
| cavke | TouchDatabase | Tool | 232 | 58 | 0 | 0 | - | 0.048 |
| qzeua | TouchDevelop Jr. | Tool | 64 | 49 | 1 | 0 | 0 | 0.029 |
| whpgc | Vulcanization calculator | Tool | 54 | 30 | 1 | 0 | 0 | 0.009 |

Figure 3: Result of the dynamic analysis of 33 TouchDevelop applications.

```
1 if (!Users.at(name).Created) {
2   Users.at(name).Created := true
3   // ...
4 }
```

The code tries to enforce a uniqueness constraint over user names, however, TouchDevelop's transactions only guarantee a causally consistent snapshot and atomic application of updates, not sufficing for the serializable execution of mixed (update and query) transactions. It is possible that two clients reserve the same name after concurrently reading `false` in line 2. A fix can be derived by reserving the name, forcing synchronization with the other clients and checking if we have won the race for the name reservation.

For example, this implementation could be used:

```
1 Users.at(name).Created := client_id
2 while (!Users.at(name).Created.confirmed)
3   sleep(0.2)
4 if (!Users.at(name).Created == client_id)
5   // ...
```

*Online Tic Tac Toe Multiplayer.* The game "Online Tic Tac Toe Multiplayer" uses "cloud game lobby", mentioned in section 1 to correctly synchronize access to games and managing participants. However, it only allows one game to be played at the same time, as moves from different games use the same data structures. In this case, we do not propose a specific fix, as the application simply does not use a correct data model.

# 7. APPLICATION: DEVELOPING CLIENTS OF WEAKLY CONSISTENT DATABASES

In this section, we show that our dynamic analysis can guide the developer while implementing an application against an eventually consistent data store. The analysis is useful when the application has consistency constraints that are not automatically guaranteed by the data store. In our experiment, our analysis always reported violations that lead to real synchronization problems (or missing lightweight specification). Moreover, the analysis correctly classifies all of our fixes as serializable.

In our case study we use Riak [13], a distributed key-value data store based on a similar design as Amazon's Dynamo [7]. Riak replicates data across a cluster of nodes and keeps it eventually consistent. Operations are typically performed in a highly available manner, where queries only contact a subset of the replicas, and updates return to the client before being confirmed by all nodes. To resolve update-update conflicts in a convergent manner, Riak provides implementations of several conflict-free replicated data types [23] such as counters, sets, flags, maps and last-writer-wins registers.

## 7.1 Dynamic analysis of Riak-backed applications

We integrate our runtime instrumentation as a shim layer around the official Python client library of Riak. This layer serves two purposes: (1) if dynamic analysis is enabled, the layer records all executed operations of the client application to an independent database (2) it gives the developer the ability to provide lightweight specifications in addition to the purely operational API of Riak.

*Recorded information.* As in section 6, we require the knowledge of visibility vi, arbitration ar, and program order po, as well as be able to check commutativity ♯ and absorp-

tion ▷ between operations. po can be trivially determined by sequentially numbering all operations performed by the same client and recording it. To check commutativity and absorption, we record all arguments and return values of each operation.

In contrast to section 6, here visibility vi cannot be tracked using vector clocks, as Riak does not provide causal consistency, and a client operation is therefore not guaranteed to observe all of its causal predecessors. Furthermore, updates are only guaranteed to become atomically visible on a per-key basis. That is why we track visibility information for every stored value separately. Each value is embedded in a Riak-DT-Map [3], along with a set of unique identifiers of all the updates applied to the value. These identifiers correspond directly to vi edges in the DSG. Since changes to the map provide atomic visibility, a client has observed an update if and only if that update's identifier is in the set. Deletions are not performed directly, but instead, the value-embedding map also contains a flag which marks the record as deleted.

Using this instrumentation, the data stored in the database grows linearly in the number of operations performed on the database. It must therefore be noted that such an instrumentation makes sense only during testing and not during production. This restriction can be partially lifted by making further assumptions: For example, by assuming that clients remain connected to the same node, implying that the set of observed updates is monotonically increasing, one can track observed updates for each client separately and prune observed updates from the observed sets. We do not apply such a technique in our evaluation, as short execution traces suffice for our purposes.

*Light-weight specifications.* If the dynamic analysis is used without any developer annotations, every operation will be observed as single-operation transaction. In that case, ECRacer essentially checks for sequential consistency [15] of the recorded execution. Our client library provides two ways of expressing the developer's intent: (1) one may designate that a set of operations forms a transaction, that is, expected to have *serializable* behavior; (2) the developer may *exclude* query operations from the serializability checking. We then allow such operations to return inconsistent values.

*Offline analysis.* The offline analysis is performed in the same manner, and in fact, with the same core implementation as in Section 6, despite that it targets different systems. ECRacer is extended with commutativity and absorption specifications for all operations provided by Riak. Here, we use the semantics of Riak's CRDTs to derive the arbitration order. For example: for an increment-only counter, all updates are unordered as they all commute, therefore the arbitration order is empty. For an add-wins set, concurrent adds are unordered, concurrent removes are unordered, and every add is ordered after all concurrent removes. Finally, for a last-writer-wins register, all updates to the register are totally ordered by their physical timestamps.

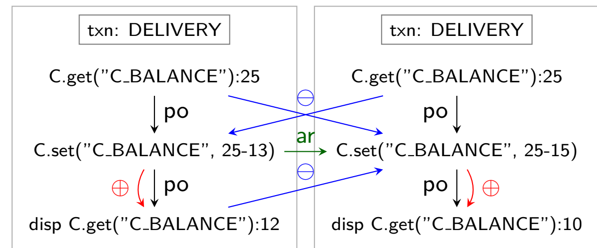
## 7.2 Implementing TPC-C using ECRacer

TPC-C [29] is one of the most well-known database benchmarks. It defines a database-backed whole-sale supplier application, featuring among others payment, delivery, order status, stock level status, and order creation transactions. It is typically implemented by vendors of databases which provide serializable transactions, but has also extensively been used for the benchmarking of weakly synchronized distributed databases [1].

We use our analysis to derive a correctly synchronized version of a TPC-C implementation, by iteratively eliminating violations detected by our analysis. Initially, we start with an implementation of TPC-C (in Python), loosely based on the sample programs given in Appendix A of the TPC-C specification [29]. These programs use a standard table-based data model and assume support for serializable transactions from the database.

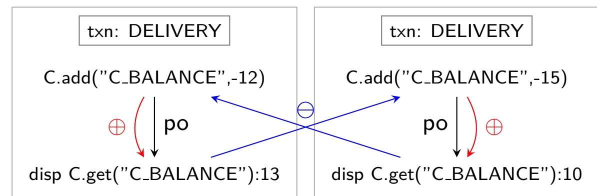
Each version of the implementation is run with the previously described runtime instrumentation for 20 seconds with 3 clients in parallel on a minimal three node setup of Riak on a remote server. The number of transactions and operations executed, the analysis time and the detected violations are listed in Figure 4.

*Version 1.* In the first version, the analysis detects 9 violations. 3 of those are due to increments being performed in a non-atomic way:



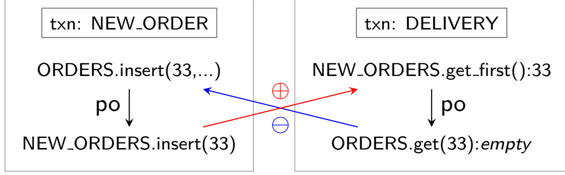

The update to the balance is lost, since Riak does not provide transactional atomicity guarantees. The problem can easily be solved without coordination, by replacing C_BALANCE by a CRDT counter with commutative increments. Note how the second part of the transactions is serializable: the left update is arbitrated before the right update, and the right update absorbs the left one. A serialization can therefore order the set/get pair sequentially to get a legal execution.

*Version 2.* The result of replacing non-atomic increments of counters by commutative counter increments, leads to the following execution fragment:




Note the difference to the previous fragment: Here, the two increments are unordered by arbitration (as they commute), and they do not absorb each other. Therefore, we get two ⊖ edges, forming a cycle in the DSG. In any serialization, one of the get queries must read the balance -2. However, since the value is only displayed to the user and has no effect on database consistency, we decide to add annotations to exclude all query operations that are merely performed for displaying data on the terminal from the analysis.

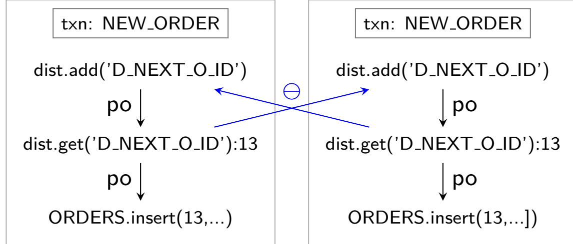
*Version 3.* In Version 3, ECRACER detects 2 violations. 1 of those is due to partial observation of transactions, as in the following cycle between NEW_ORDER and DELIVERY:




The left transactions inserts a new order, and inserts a foreign key to that order into the NEW_ORDERS table. The right transaction observes the foreign key, but does not observe the corresponding order record. The DSG defines that, in a serialization, the order insertion must follow the order retrieval to make its return value legal, but also requires the foreign key insertion to be ordered before the foreign key retrieval, creating a cycle with the program order.

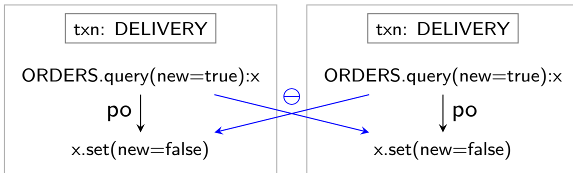
We can solve this and similar errors by denormalizing data and combining tables into nested data structures. In this example, we embed the NEW_ORDER flag into the ORDERS record. Similarly, the lines of the order are embedded as a set CRDT in the ORDERS record.

*Version 4.* In Version 4 of the implementation, only one violation is detected: Two parallel increments to D_NEXT_O_ID (the district's next, serially assigned order number) and two queries to that value in the same NEW_ORDER transaction:




Clearly, this behavior is non-serializable, as it should not be possible for both transaction instances to read value 13 in the second operation. This problem is classic for TPC-C and it was previously shown to be impossible to implement without coordination [1]. To resolve the problem (which is not directly possible in RIAK), we use atomic counters, externally synchronized by ZooKeeper [12], a high-performance service for distributed synchronization.

*Version 5.* Finally, after running the Version 4 several times, the analysis reported potential double delivery, a rare circumstance due to the infrequent execution of the delivery transaction:




Here, the transactions receives all new orders from the database, and subsequently disables the new flag. Behaving correctly under serializable guarantees, this implementation may lead to double deliveries. We solve the problem by exploiting the rarity of the delivery transactions: We can force its execution on a single server and lock it locally, without compromising performance.

| Version | #Txns | #Ops | Time [s] | #Viol. |
|---|---|---|---|---|
| 1 | 280 | 7197 | 28.064 | 9 |
| 2 | 307 | 6907 | 24.806 | 6 |
| 3 | 365 | 7236 | 20.938 | 2 |
| 4 | 441 | 6903 | 7.678 | 1 |
| 5 | 475 | 7192 | 8.113 | 1 |
| 6 | 449 | 6903 | 6.823 | 0 |

Figure 4: Analysis results for each version of TPC-C. **#Txns** is the number of transactions, **#Ops** the number of operations recorded, **Time** is the total analysis time in seconds, **#Viol.** are the number of detected violations

*Version 6.* In the final version, no serializability violation is detected. While ECRACER, being a dynamic analysis, cannot provide a guarantee about the absence of violations, one can gain significant confidence by creating bad-case scenarios (network partitions, node failure, etc.) during the dynamic analysis.

## 7.3 Scalability

Finally, we evaluate the throughput of our incrementally derived, partially serializable implementation to an implementation with straight-forward synchronization. The former, labeled *Custom* in Figure 5 corresponds to Version 6 from the previous subsection, while the latter, labeled *Locked*, corresponds to Version 1, extended with an uninformed attempt at correct synchronization: Clients lock parts of the database that they access using ZooKeeper primitives.

We run both version on 4 to 10 `m4-large` Amazon EC2 instances with 2 virtual cores and 8GiB of RAM each, running clustered instances of both RIAK an ZooKeeper. RIAK is run in its default configuration with triple replication and Apache SOLR providing advanced querying on top of the key-value store.

Our benchmark follows standard usage of distributed databases: it replicates data across nodes for failure tolerance, and it does not use stored procedures to implement transactions. It is therefore not directly comparable to optimized implementations, and much higher throughputs can be achieved with further domain knowledge. However, the benchmark clearly shows that the manual synchronization derived from our analysis result, without any domain knowledge, scales much better then the uninformed locking approach.

# 8. CONCLUSION

We presented a new serializability criterion for eventually consistent data stores. The criterion generalizes the classic notion of conflict serializability to deal with high level data types by leveraging the concepts of commutativity and absorption.

We built a prototype dynamic analyzer for detecting violations of our criterion and evaluated it on both mobile applications that use weak-replication as well as a well-known database benchmark. Our experimental results indicate that the criterion is practically useful: it captures one's intuitive

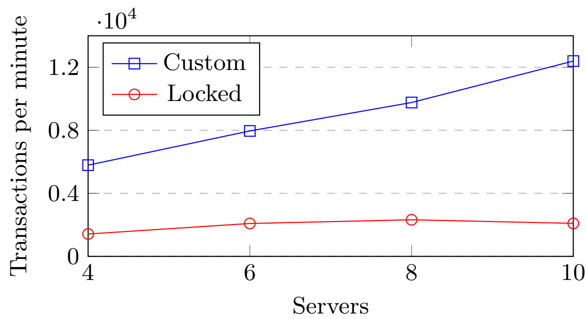



Figure 5: Performance comparison of Version 6 (*Custom*) and a primitively synchronized variant of Version 1 (*Locked*)

notion of correctness in this setting and led to finding several previously undetected bug.

As a result, we believe the concepts and systems presented in this work are a useful step forward in helping build correct and efficient applications on top of eventually consistent data stores.

## 9. REFERENCES


[1] P. Bailis, A. Fekete, M. J. Franklin, A. Ghodsi, J. M. Hellerstein, and I. Stoica. Coordination avoidance in database systems. *Proc. VLDB Endow.*, 8(3):185–196, Nov. 2014.

[2] A. Bouajjani, E. Derevenetc, and R. Meyer. Checking and enforcing robustness against tso. In *Proceedings of the 22Nd European Conference on Programming Languages and Systems*, ESOP'13, pages 533–553, Berlin, Heidelberg, 2013. Springer-Verlag.

[3] R. Brown, S. Cribbs, C. Meiklejohn, and S. Elliott. Riak dt map: A composable, convergent replicated dictionary. In *Proceedings of the First Workshop on Principles and Practice of Eventual Consistency*, PaPEC '14, pages 1:1–1:1, New York, NY, USA, 2014. ACM.

[4] S. Burckhardt, A. Gotsman, H. Yang, and M. Zawirski. Replicated data types: Specification, verification, optimality. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, pages 271–284, New York, NY, USA, 2014. ACM.

[5] S. Burckhardt, D. Leijen, J. Protzenko, and M. Fähndrich. Global sequence protocol: A robust abstraction for replicated shared state. In J. T. Boyland, editor, *29th European Conference on Object-Oriented Programming*, volume 37 of *ECOOP 2015*, pages 568–590, Dagstuhl, Germany, 2015. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[6] A. Cerone, G. Bernardi, and A. Gotsman. A Framework for Transactional Consistency Models with Atomic Visibility. In L. Aceto and D. de Frutos Escrig, editors, *26th International Conference on Concurrency Theory*, volume 42 of *CONCUR 2015*, pages 58–71, Dagstuhl, Germany, 2015. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[7] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 205–220, New York, NY, USA, 2007. ACM.

[8] D. Dimitrov, V. Raychev, M. Vechev, and E. Koskinen. Commutativity race detection. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 305–315, New York, NY, USA, 2014. ACM.

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

[10] G. Gierz, K. Hofmann, K. Keimel, J. Lawson, M. Mislove, and D. Scott. *Continuous Lattices and Domains*. Number 93 in Encyclopedia of Mathematics and its Applications. Cambridge University Press, 2003.

[11] S. Gilbert and N. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, June 2002.

[12] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'10, pages 11–11, Berkeley, CA, USA, 2010. USENIX Association.

[13] R. Klophaus. Riak core: Building distributed applications without shared state. In *ACM SIGPLAN Commercial Users of Functional Programming*, CUFP '10, pages 14:1–14:1, New York, NY, USA, 2010. ACM.

[14] A. Lakshman and P. Malik. Cassandra: A decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, Apr. 2010.

[15] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comput.*, 28(9):690–691, Sept. 1979.

[16] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Don't settle for eventual: Scalable causal consistency for wide-area storage with cops. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 401–416, New York, NY, USA, 2011. ACM.

[17] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Stronger semantics for low-latency geo-replicated storage. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, nsdi'13, pages 313–328, Berkeley, CA, USA, 2013. USENIX Association.

[18] F. Mattern. Virtual time and global states of distributed systems. In C. M. et al., editor, *Proc. Workshop on Parallel and Distributed Algorithms*, pages 215–226, North-Holland / Elsevier, 1989. (Reprinted in: Z. Yang, T.A. Marsland (Eds.), "Global States and Time in Distributed Systems", IEEE, 1994, pp. 123-133.).

[19] A. Mazurkiewicz. Trace theory. In W. Brauer, W. Reisig, and G. Rozenberg, editors, *Petri Nets: Applications and Relationships to Other Models of Concurrency*, volume 255 of *Lecture Notes in Computer Science*, pages 278–324. Springer Berlin Heidelberg, 1987.

[20] S. Owens. Reasoning about the implementation of concurrency abstractions on x86-tso. In *Proceedings of the 24th European Conference on Object-oriented*

*Programming*, ECOOP'10, pages 478–503, Berlin, Heidelberg, 2010. Springer-Verlag.

[21] C. H. Papadimitriou. The serializability of concurrent database updates. *J. ACM*, 26(4):631–653, Oct. 1979.

[22] N. Preguiça, M. Zawirski, A. Bieniusa, S. Duarte, V. Balegas, C. Baquero, and M. Shapiro. Swiftcloud: Fault-tolerant geo-replication integrated all the way to the client machine. In *Proceedings of the 2014 IEEE 33rd International Symposium on Reliable Distributed Systems Workshops*, SRDSW '14, pages 30–33, Washington, DC, USA, 2014. IEEE Computer Society.

[23] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. A comprehensive study of Convergent and Commutative Replicated Data Types. Research Report RR-7506, Inria – Centre Paris-Rocquencourt ; INRIA, Jan. 2011.

[24] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. Conflict-free replicated data types. In *Proceedings of the 13th International Conference on Stabilization, Safety, and Security of Distributed Systems*, SSS'11, pages 386–400, Berlin, Heidelberg, 2011. Springer-Verlag.

[25] D. Shasha and M. Snir. Efficient and correct execution of parallel programs that share memory. *ACM Trans. Program. Lang. Syst.*, 10(2):282–312, Apr. 1988.

[26] S. Sivasubramanian. Amazon dynamodb: A seamlessly scalable non-relational database service. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 729–730, New York, NY, USA, 2012. ACM.

[27] Y. Sovran, R. Power, M. K. Aguilera, and J. Li. Transactional storage for geo-replicated systems. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 385–400, New York, NY, USA, 2011. ACM.

[28] N. Tillmann, M. Moskal, J. de Halleux, and M. Fahndrich. Touchdevelop: Programming cloud-connected mobile devices via touchscreen. In *Proceedings of the 10th SIGPLAN Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! 2011, pages 49–60, New York, NY, USA, 2011. ACM.

[29] T. P. P. C. (TPC). Tpc benchmark c, revision 5.11, Feb. 2010.

[30] M. Vaziri, F. Tip, and J. Dolby. Associating synchronization constraints with data in an object-oriented language. In *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '06, pages 334–345, New York, NY, USA, 2006. ACM.

[31] W. E. Weihl. Commutativity-based concurrency control for abstract data types. *IEEE Trans. Comput.*, 37(12):1488–1505, Dec. 1988.

[32] K. Zellag and B. Kemme. How consistent is your cloud application? In *Proceedings of the Third ACM Symposium on Cloud Computing*, SoCC '12, pages 6:1–6:14, New York, NY, USA, 2012. ACM.