IPA: Invariant-Preserving Applications for Weakly Consistent Replicated Databases

Valter Balegas
NOVA LINCS, FCT,
Universidade NOVA de Lisboa
v.sousa@campus.fct.unl.pt

Sérgio Duarte
NOVA LINCS, FCT,
Universidade NOVA de Lisboa
smd@fct.unl.pt

Carla Ferreira
NOVA LINCS, FCT,
Universidade NOVA de Lisboa
carla.ferreira@fct.unl.pt

Rodrigo Rodrigues
INESC-ID, Instituto Superior
Técnico, U. Lisboa
rodrigo.rodrigues@inesc-id.pt

Nuno Preguiça
NOVA LINCS, FCT,
Universidade NOVA de Lisboa
nuno.preguica@fct.unl.pt

ABSTRACT

It is common to use weakly consistent replication to achieve high availability and low latency at a global scale. In this setting, concurrent updates may lead to states where application invariants do not hold. Some systems coordinate the execution of (conflicting) operations to avoid invariant violations, leading to high latency and reduced availability for those operations. This problem is worsened by the difficulty in identifying precisely which operations conflict.

In this paper we propose a novel approach to preserve application invariants without coordinating the execution of operations. The approach consists of modifying operations in a way that application invariants are maintained in the presence of concurrent updates. When no conflicting updates occur, the modified operations present their original semantics. Otherwise, we use sensible and deterministic conflict resolution policies that preserve the invariants of the application. To implement this approach, we developed a static analysis, IPA, that identifies conflicting operations and proposes the necessary modifications to operations.

Our analysis shows that IPA can avoid invariant violations in many applications, including typical database applications. Our evaluation reveals that the offline static analysis runs fast enough for being used with large applications. The overhead introduced in the modified operations is low and it leads to lower latency and higher throughput when compared with other approaches that enforce invariants.

1. INTRODUCTION

Databases are commonly replicated in different geographical regions to ensure low latency and high availability across the globe [22, 49, 20]. To provide these guarantees, systems often adopt weak consistency. This approach can expose temporary state divergence to clients, making application development more difficult. Many techniques have been proposed to make these systems easier to program: conflict-free replicated data types [45] (CRDTs) ensure state convergence; causal consistency [39, 12, 57] enforces that operations are made visible respecting the happens-before relation; and, finally, highly available transactions [8, 11, 40, 49, 57] can group multiple operations that must take effect all at once.

Despite these proposals, it remains difficult to develop applications under weak consistency. Several studies [10, 32, 53] show that, in many applications, concurrent executions lead to the violation of application invariants, resulting in inconsistent states. To give an example, consider an online e-games platform, and assume that the application is correct when executed under strong consistency. Consider that a user enrolls in some tournament and concurrently the organizer decides to cancel the tournament. The first operation creates a new reference between the user and the tournament, while the second removes the tournament, and all references to it. Since these operations are concurrent, the remove operation does not delete the newly created reference, which now points to a tournament that no longer exists, thus breaking referential integrity.

To prevent invariant violations efficiently, some systems provide primitives for executing operations synchronously [49, 37], paying the cost of coordination only when necessary. However, it is difficult to identify the problematic executions, especially because inconsistencies may occur only for some combinations of operations and in some specific states. The result is that programmers often end up constraining concurrency too much to preserve correctness, with an impact on the availability and latency of systems [14, 19].

This paper proposes a novel approach for preserving application invariants under weak consistency that does not impact the availability and latency of applications. The key idea is to extend operations with updates that preventively guarantee the preservation of invariants in the presence of concurrent updates. The additional updates should have no visible effect if no concurrent operation is executed, keeping the semantics of the operation in the sequential case. These updates come into effect when needed to correct the undesirable semantics of concurrent operations. In our previous example, it is possible to maintain referential integrity by restoring the removed tournament. To this end, the enroll operation is extended with an update that prevents the concurrent deletion of the tournament.

To help programmers adopt our approach, we propose a methodology for modifying applications. The key element of the methodology is our invariant-preservation analysis (IPA) and static analysis tool that relies on information about the application, including invariants and operations, to identify which operations might lead to invariant violations and to suggest modifications to the operations to prevent those violations from occurring. Previous work employed static analysis to optimize the use of coordination in applications running under weak consistency [36, 14, 44]. In contrast, instead of resorting to coordination for avoiding the concurrent execution of conflicting operations, we allow operations to execute concurrently and leverage conflict resolution policies to ensure a result that is deterministic (given the operations that execute concurrently) and preserves invariants. To our knowledge, our work is the first to adopt such an approach to enforce invariants that span multiple database objects (that can be stored on different machines).

Our evaluation is threefold. First, we analyzed a number of applications/benchmarks that are representative of common OLTP applications, concluding that our approach can identify the most common relational database invariant violations, and that the suggested modifications lead to sensible and deterministic semantics in the concurrent case. Furthermore, the semantics of the modified operations is equivalent to the original operations when no conflicting operation executes concurrently. Second, we analyze the scalability of the static analysis process, showing that it is fast enough to be used with large applications. Third, the results of our experiments show that the performance of the modified applications has only a small overhead compared to the unmodified versions and is faster than a state-of-the-art solution that maintains invariants using coordination to prevent invariant violations.

In this paper we make the following contributions: (i) a novel approach to preserve invariants under weak consistency without resorting to coordination, which combines the extension of operations with new updates and the use of appropriate conflict resolution policies; (ii) an algorithm that takes information about operations and application invariants and proposes modifications to operations and conflict resolution policies to preserve invariants while keeping the original semantics of the operations in the absence of conflicts; (iii) the design of new CRDTs that support the conflict resolution policies necessary for adopting our approach; (iv) an implementation and evaluation of the proposed approach.

The paper discusses the difficulties in designing applications on top of weak consistency (§2); presents an overview of the IPA approach (§3); presents the system model and defines the key principles for IPA (§4); details the IPA analysis (§5); discusses implementation details (§6); evaluates the approach and prototype (§7); discusses related work (§8); and presents some final remarks (§9).

2. BACKGROUND

In this section we discuss the problem of application correctness when executing applications under weak consistency.

2.1 Eventual consistency and CRDTs

Storage systems that adopt weak consistency models, such as eventual consistency [50, 52, 22, 34] or causal+ consistency [39, 40, 57], need to include a mechanism to merge concurrent updates, guaranteeing that all replicas converge to the same state after applying the same set of updates. In *last-writer-wins*, the latest update, according to some total order defined among updates, prevails. This strategy may lead to lost updates, as the effects of an update may be overwritten by a concurrent update. To address this problem, some systems, such as Cassandra, support type-specific merge for some data types – e.g., for counters, the final value reflects all updates.

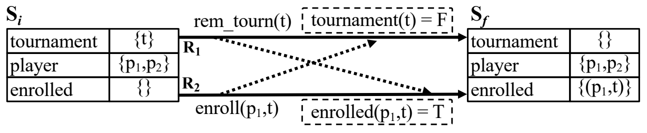


Figure 1: Concurrent execution of $enroll(p, t)$ and $rem\_tourn(t)$ leads to an invariant violation.

Conflict-free replicated data types (CRDTs) [45, 46] are a principled approach for defining replicated objects. A CRDT is an abstract data type designed to be replicated. Any replica of a CRDT can be modified without coordination and any two replicas that receive the same set of updates converge to the same state, deterministically.

A CRDT implements type-specific *concurrency semantics*, namely defining how to merge concurrent updates. For example, a *set* exports two operations for adding and removing an element: $add(e)$ and $remove(e)$. As these operations do not commute when referring to the same element, the concurrency semantics must arbitrate the state of the object when an $add(e)$ and a $remove(e)$ execute concurrently. Several *concurrency semantics* are possible. In the *Add-wins* semantics, an add wins (takes priority) over a concurrent remove, leading to a state where e belongs to the set after applying both updates. In the *Rem-wins* semantics, the remove wins over a concurrent add, leading to a state where e does not belong to the set.

CRDTs were first used in research systems, such as Walter [49] and SwiftCloud [57]. More recently, they were adopted in production systems, such as Redis [18], Akka [4] and Riak [17].

2.2 Convergence is not enough

State convergence is not sufficient to guarantee application correctness, as the final state might be invalid. This can occur when the rules for maintaining convergence are defined per data-type, not considering the relations among the multiple objects of the state.

We illustrate this problem with the application that stores information for an e-games platform introduced before. In this application, the database stores information about $players$, $tournaments$ and which players are *enrolled* in which tournaments. Players can enroll or disenroll from tournaments, and a tournament can be removed by the administrator if there are no players enrolled. There is an implicit referential integrity invariant that states that a player may only enroll in an existing tournament.

Now consider the concurrent execution of two operations to remove a tournament and enroll a player in that tournament, as shown in Figure 1. In state S_i, the remove verifies that there is no entry in the enrolled table referring to tournament t and removes it. In the same state, the enroll operation creates a new reference between player p_1 and t. Since the effects of each operation are generated in different replicas, the remove operation does not see p_1 enrolled in t, whereas enroll still sees t. When the state of both replicas converges (S_f), the state will have player p_1 enrolled in tournament t, which no longer exists, thus breaking referential integrity.

We note that this problem still occurs in weakly consistent systems that provide additional guarantees, such as causal+ consistency and highly available transactions. Causal+ consistency [39, 40, 6] guarantees that the effects of an operation are only visible after the effects of all operations that happened-before [35] it. However, this does not impact the execution of concurrent operations.

Some systems [8, 11, 57, 40] provide highly available transactions, in which a set of updates is applied atomically in a replica: concurrent transactions either see the effects of all updates or none. This type of transactions differs from ACID transactions by allowing concurrent updates to the same object, and the final value of the object is defined by the conflict resolution policy of the object.

3. OVERVIEW

In this section we explain our proposal to modify applications to ensure invariant preservation without coordination.

The invariant violation of Figure 1 can be repaired, after it is detected, by either: (i) removing the new player enrollment from the enrolled table; or (ii) restoring the tournament to its previous state. In this type of approach, known as compensations [50, 27], when the system detects that the database is inconsistent, it applies some compensation effects to restore the database integrity.

Our insight is that in many situations the effects to restore the database integrity can be applied preventively alongside the original operations, repairing the invariant violation automatically in a conflicting execution. Going back to our previous example, restoring a tournament to its previous state can be achieved by executing a touch operation in the tournament when executing the enroll, and adopting a conflict resolution policy where the touch wins over a concurrent delete. The touch operation has no observable effect, only updating the metadata to guarantee that the concurrent execution is detected and solved according to the defined conflict resolution policy. This approach has several interesting properties. First, it does not require any form of coordination during the execution of operations, or any mechanism for detecting invariant violations at runtime (which can be expensive, particularly when the invariant relates data stored in different servers). Second, the additional operations have no observable effect if no conflicting concurrent operations are executed.

This approach requires combining appropriate conflict resolution policies with a careful selection of which updates to add to each operation. The goal is to guarantee that the extra updates have no observable effect unless a conflicting operation is executed, and that, in such cases, the additional updates guarantee that the invariant is preserved. Additionally, it is necessary to guarantee that the extra updates do not lead to the violation of any other invariant and that modified operations do not interfere among themselves.

To help in this process, we devised algorithms that use static analysis to detect executions that might violate an application invariant, and search for modifications that prevent those violations. Typically, there will be several alternative modifications for preserving invariants – the programmer must select the most appropriate for her application. In most cases, the additional updates have no observable effect and can be applied preventively with the operations.

For some cases, the additional updates might have an observable effect on the database state. To address these cases, we also support a compensation mechanism that applies these updates if a conflict violation is detected. For instance, it is not possible to prevent flight overbooking, but it is possible to compensate the event by reimbursing users or finding alternative flights. Unlike other solutions, our approach does not require coordination to execute compensations, which can execute at any replica. However, the detection of invariant violation is limited to conditions over the state of a single object.

Example: We now detail how our approach works, using the example of Figure 1. Consider that the database tables, *player* and *tournament*, and relationship, *enrolled*, are stored in separate sets.

As mentioned before, for preventing the invariant violation by recreating the deleted tournament, it suffices to extend the effects of the enroll operation to *touch* tournament t. By adopting the *Add-wins* policy for the tournament, with the touch tournament winning over a concurrent remove tournament, this guarantees that tournament t survives the concurrent execution of a remove t operation.

For preventing the invariant violation by deleting enrollments in removed tournaments, it suffices to extend the effects of the remove tournament t to preventively *remove* any concurrently enrolled pair associated with t. This does not produce any observable effect because there should be no element enrolled in t after the execution of the remove operation. Adopting a *Rem-wins* policy for the *enrolled* set guarantees that a concurrent *enroll* will have no effect.

We can view the modifications to the operations as a way of giving priority to one conflicting operation over the other, the same way that a *Add-wins* (resp. *Rem-wins*) set CRDT gives priority to an add over a remove (resp. a remove over an add). The additional updates guarantee that the preconditions for executing the operation that is given priority remain valid despite the execution of any concurrent operation. For example, recreating the tournament is to give priority to the enroll over the remove tournament. The enroll precondition that would be violated is that the tournament exists. Touching the tournament and selecting the *Add-wins* policy guarantees that the precondition remains valid despite concurrent remove tournaments.

4. IPA APPROACH

This section introduces the main principles underlying IPA.

4.1 System model

We consider a database composed of a set of objects fully replicated in multiple data centers. Operations over those objects can execute a sequence of reads and updates enclosed in a transaction. As the transaction executes in an initial replica, the effects of updates are recorded and queued for replication upon transaction commit. Propagation of updates can be asynchronous and must respect causal order. Hereafter, we use the term operation to refer updates produced by the execution of the transaction code in the initial replica.

We denote by $o(S)$ the state after applying the updates of operation o to state S. A database snapshot, S_n, is the state of the database after executing a sequence of operations $o_1, \ldots, o_n$ in the initial database state, S_{init}, i.e., $S_n = o_n(\ldots(o_1(S_{init})))$. The set of operations reflected in snapshot S is denoted by $Ops(S)$, e.g., $Ops(S_n) = \{o_1, \ldots, o_n\}$. The state of a replica results from applying both local and remote operations, in the order received.

We say that an operation o_a happened-before [35] operation o_b, executed initially in database snapshot S_b, $o_a \prec o_b$, iff $o_a \in Ops(S_b)$. Operations o_a and o_b are concurrent, $o_a \parallel o_b$ iff $o_a \not\prec o_b \wedge o_b \not\prec o_a$.

For an execution of a given set of operations O, the happens-before relation defines a partial order among operations, $\mathcal{O} = (O,\prec)$. We say $\mathcal{O}' = (O,<)$ is a valid serialization of $\mathcal{O} = (O,\prec)$ if $\mathcal{O}'$ is a linear extension of $\mathcal{O}$, i.e., $<$ is a total order compatible with $\prec$.

Operations can execute concurrently, with each replica executing operations according to a different valid serialization. To guarantee state convergence, we assume the system gives the programmer the choice of various deterministic conflict resolution policies on a per-object basis, i.e., the result of applying updates that were executed concurrently is deterministic independently of the execution order. In our prototype, we rely on CRDTs [45, 49] to achieve this goal.

We consider that application correctness can be expressed in terms of invariants [9, 14, 29]. An invariant is a logical condition expressed over the database state. A given state S preserves an invariant I iff $I(S) = true$, where $I(S)$ is a function that checks the validity of the invariant in state S. A state S_i is *I-valid* (or simply valid) iff $I(S_i) = true$; otherwise the state is *I-invalid* (or simply invalid). We require the initial state, S_{init}, to be valid.

We say that $\mathcal{O}' = (O,<)$ is an I-valid serialization of $\mathcal{O} = (O,\prec)$ if $\mathcal{O}'$ is a valid serialization of $\mathcal{O}$, and I holds in every state that results from executing any possible prefix of $\mathcal{O}'$. If I is the conjunction of all application invariants, then we say that an application is correct if, in any possible execution of that application, every replica evolves through a sequence of I-valid states. We say that an operation o_1 conflicts with o_2 if the execution of o_1 makes the preconditions of o_2 false in some database state.

4.2 Principles for IPA

We now present the key principles for guaranteeing the correct execution of an application under weak consistency. We follow the notions introduced by Bailis et al. [9], adapting them to our model.

DEFINITION 1. *Given a set of commutative operations O and the happens-before relation, $\prec$, we say O is* I-Confluent *[9] iff any state S, obtained by executing a prefix of any valid serialization of $(O, \prec)$, starting from an* I-valid *state, is* I-valid*.*

This means that for a set of *I-Confluent* operations, despite executing operations in a different serialization order, every replica will evolve only through *I-valid* states. Along with the commutativity of the operations, this guarantees the correctness of application execution both in terms of convergence and invariant-preservation.

To preserve invariants, an operation should only produce side effects in states that satisfy the operation preconditions. For example, for adding a player to a tournament, the player and tournament must exist. When an operation executes in the initial replica, the code of the operation verifies that the local state satisfies the preconditions.

The challenge arises when operation side-effects propagate asynchronously to remote replicas. At the remote replica, concurrent operations may have already executed, leading to a state where the operation preconditions do not hold anymore. Applying the side-effects as-is may result in an invariant violation – e.g., applying the effects of adding a player to a tournament in a state where the tournament has been removed leads to an invariant violation.

DEFINITION 2. *Given a set of operations O and the happens-before order, $\prec$, we say that S is an admissible state for $o \in O$ iff there is a valid serialization of $(O, \prec)$ in which S results from applying all operations that precede o to the initial state.*

With this definition in place, we can state a sufficient condition for having *I-Confluent* operations, thus enabling the system to execute operations in remote replicas without violating the invariants.

THEOREM 1. *Given a set of commutative operations O and the happens-before order among them, $\prec$, if for any operation o and admissible state S of o, $o(S)$ is also an* I-valid *state, then O is an* I-Confluent *set.*

The key insight of IPA is that, in many cases, it is possible to guarantee both commutativity and the sufficient property of Theorem 1 by leveraging CRDTs and extending operations with updates that restore the operation preconditions. In the example of the previous section, an operation to enroll a player in a tournament can execute safely if it restores the player and tournament. This can be achieved by touching both the player and tournament and using an *Add-wins* conflict resolution policy for players and tournaments, thus protecting the enroll operation against concurrent removal of the player or tournament. The deterministic nature of *Add-wins* and *Rem-wins*, with conflicts solved based only on the type of concurrent operations, is key for achieving the intended result.

We note that it is only necessary to restore the preconditions in admissible states, as the serialization of operations must be consistent with the happens-before relation. In practice, it is necessary to execute operations in causal order and revert the effects of concurrent updates that may affect the preconditions of the operation. The additional updates in an operation should be executed atomically with the updates of the operation to guarantee that no inconsistency is observed. In our prototype, we achieve this by relying on highly available transactions [8, 57]. In the next sections we show how to put these principles into practice.

```
// Application invariants
@Inv("forall(Player:p, Tournament:t) :-
       enrolled(p, t) ==> player(p) and tournament(t)")
@Inv("forall(Player:p,q, Tournament:t) :- inMatch(p, q, t) ==>
       enrolled(p,t) and enrolled(q,t) and (active(t) or finished(t))")
@Inv("forall(Tournament:t) :- #enrolled(*, t) <= Capacity")
@Inv("forall(Tournament:t) :- active(t) or finished(t) ==> tournament(t)")
@Inv("forall(Tournament:t) :- not( active(t) and finished(t))")

public interface TournamentApp { // Operations effects and signatures
 @True("player(p)")
 RESULT add_player(Player p);

 @False("player(p)")
 RESULT rem_player(Player p);

 @True("tournament(t)")
 RESULT add_tourn(Tournament t);

 @False("tournament(t)")
 RESULT rem_tourn(Tournament t);

 @True("enrolled(p, t)")
 @Inc("#enrolled(*, t)")
 RESULT enroll(Player p, Tournament t);

 @False("enrolled(p, t)")
 @Dec("#enrolled(*, t)")
 RESULT disenroll(Player p, Tournament t);

 @True("active(t)")
 RESULT begin_tourn(Tournament t);

 @True("finished(t)")
 @False("active(t)")
 RESULT finish_tourn(Tournament t);

 @True("inMatch(p, q, t)")
 RESULT do_match(Player p, Player q, Tournament t);
}
```

Figure 2: *Tournament* application specification.

5. IPA DESIGN

We now present our methodology for developing *I-Confluent* applications, comprising the following three steps.

Step 1: Specification: The first step consists of building a specification of the application by identifying application invariants and operation effects. We use the same specification language used in Indigo [14], requiring programmers to specify the invariants and the effects of each operation using first-order logic.

Step 2: IPA analysis: The IPA analysis, performed by our tool, is an iterative process where, in each iteration: (i) the tool identifies a pair of conflicting operations, i.e., operations that might break some invariant when executed concurrently, and proposes modifications that guarantee that invariants are preserved; (ii) the programmer chooses which conflict resolution he or she prefers. This process executes multiple times until no more conflicts exist.

Step 3: Code modification: The analysis returns a new specification of the application, which contains the selected modifications, comprising both the use of appropriate conflict resolution policies for each object and the modification to operations to avoid invariant violations. When the modifications produce no observable effect, they are appended to the corresponding operations. Otherwise, they must execute as compensations when a conflict occurs.

Fully patched applications can then execute in any replicated system that provides causal consistency, highly available transactions and the necessary type-specific conflict resolution policies. A number of systems support these features [49, 57, 5].

5.1 Specification

The specification of an application conveys information about invariants and operation effects. This is done using first-order logic, which is sufficiently expressive to cover most common relational databases constraints and operation effects [14, 29, 10].

In Figure 2, we present the specification of the tournament application. Predicates are used to represent the database state. Invariants are represented by quantified boolean statements, and operation effects are modeled with predicate assignments. For example, the logical implication in line 2 specifies an invariant that says that a player p may only be enrolled in a tournament t if player p and tournament t exist in the database. Predicate assignments can either set the value of a boolean predicate to $true$ or $false$, or apply an arithmetic operator to the predicate, in the case of numerical predicates. As an example, the effects of operation $enroll(p, t)$ set the predicate $enrolled(p, t)$ to true (line 21) and increment the number of elements in the enrolled set for tournament t (line 22).

5.2 IPA analysis

The IPA analysis is an iterative process to modify the operations of an application in order to guarantee that application invariants are preserved when operations execute concurrently.

Algorithm 1 presents the algorithm for executing the IPA analysis. The analysis builds on two main components: (i) *conflict detection*, for identifying pairs of operations that may cause an invariant violation; and (ii) *conflict repair*, for finding a modification to a pair of conflicting operations that makes them invariant-preserving.

The main function (line 1) has three parameters: (1) the invariant, I, which is a single expression that connects all invariants with a conjunction operator; (2) the set of operations, Ops, with all operations defined in the application; and (3) the initial conflict-resolution policies for the predicates, CR, as defined by the programmer. The function consists of a loop that finds pairs of conflicting operations and a repair for each conflict. The repair consists of an extended version of the operations, which replaces the original ones, and appropriate conflict resolution policies for predicates. The loop continues until no more conflicting operations exist.

Next, we detail the main components of the analysis. For simplicity, our presentation omits some details. First, it ignores the situation when no repair can be found, which is discussed in Section 5.5. Second, the algorithm only handles boolean predicates. We discuss numerical invariants and compensations in Section 5.3.

5.2.1 *Conflict detection*

The conflict detection algorithm finds a pair of operations that, when executed concurrently, may break the application invariants. To this end, it considers all pairs of operations in the specification (checking conflicts pairwise is sound, as shown independently by Gotsman [29] and Balegas [48]).

For each pair, the algorithm first checks if the operations have opposing effects (line 8), i.e., if one sets a predicate to true while the other sets the same predicate to false, as when $add\_tourn(t)$ and $rem\_tourn(t)$ execute concurrently. For each opposing effect, the algorithm uses the rule specified in CR to set the value of the predicate in the operations (line 9). If no rule exists for the predicate, the programmer is asked which rule should be used (*Add-wins* or *Rem-wins*), and the updated CR rules are returned by the function.

For checking if the concurrent execution of both operations may break an invariant, it is necessary to consider their concurrent execution in all valid states. If any resulting state does not respect I, then the operations conflict. To search for an invalid execution efficiently, we rely on an SMT solver (line 10), which uses several optimizations and heuristics to avoid testing all cases exhaustively.

Figure 3a exemplifies the conflict detection procedure for two operations, $rem\_tourn(t)$ and $enroll(p, t)$. To select the states to be checked, the SMT solver determines, from the invariants, the weakest precondition for executing both operations. In this case, it is necessary that $tournament(t)$ and $player(p)$ are set to true.

Algorithm 1 IPA algorithm and main functions.

```
    ▷ IPA main loop.
 1: function IPA($I$, $Ops$, $CR$)
 2:     while existsConflictingPair($I$, $Ops$, $CR$) do
 3:         $opPair$ ← findConflictingPair($I$, $Ops$, $CR$)
 4:         ($newPair$,$CR$)← repairConflicts($I$, $opPair$, $CR$)
 5:         $Ops$.replace($opPair$, $newPair$)
 6:     return ($Ops$,$CR$)
    ▷ Checks if a pair is conflicting. [invoked in line 17]
 7: function ISCONFLICTING($I$, $OpPair$, $CR$)
 8:     if opposingEffects($OpPair$) then
 9:         ($OpPair$, $CR$) ← apply($OpPair$, $CR$)
10:     return (SMTCheckConflicting($I$, $OpPair$), $CR$)
    ▷ IPA algorithm for repairing conflicts. [invoked in line 4]
11: function REPAIRCONFLICTS($I$, $OpPair$, $CR$)
12:     $sols$ ← $\emptyset$
13:     $invPreds$ ← {getPreds($i$) | $i \in$ invClauses($I$, $opPair$)}
14:     $newOpPairsList$ ← generate($invPreds$, $I$, $OpPair$)
15:     for $newOpPair \in newOpPairsList$ do
16:         if not isPairSubset($newOpPair$, $sols$) then
17:             ($result$,$newCR$)← isConflicting($I$,$newOpPair$,$CR$)
18:             if $result == FALSE$ then
19:                 $sols$ ← $sols \cup \{(newOpPair, newCR)\}$
20:     return USERPickResolution($sols$)
    ▷ New operation generation. [invoked in line 14]
21: function GENERATE($invPreds$, $I$, ($op1$, $op2$))
22:     $seed$ ← $\{p(true), p(false) \mid p \in$ invPreds$\}$
23:     $effectSets$ ← powerSetFiltered($seed$)
24:     $pairs$ ← $\emptyset$
25:     for $p \in effectSets$ do
26:         $pairs$ ← $pairs \cup$ {(newOp($op1$, $p$),$op2$)}
27:         $pairs$ ← $pairs \cup$ {($op1$,newOp($op2$, $p$))}
28:     return order($pairs$)              ▷ by increasing no. of predicates.
```

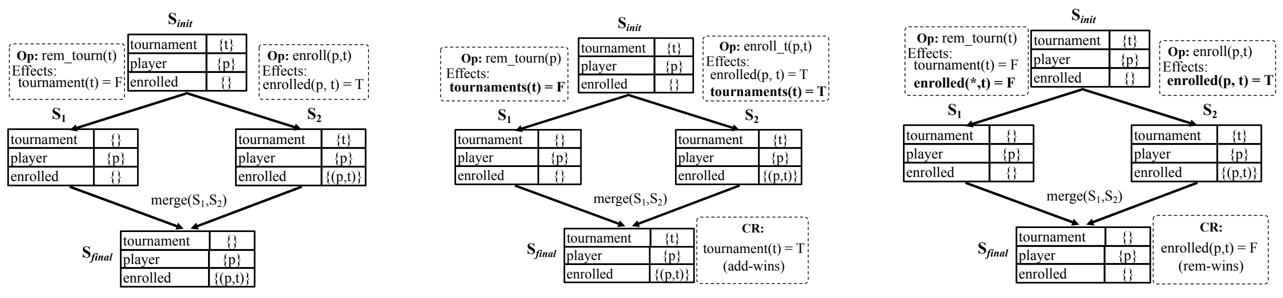
The SMT solver then checks if, for every state in which the weakest preconditions hold, the concurrent execution of both operations produces a valid state. In our example, executing $rem\_tourn(t)$ and $enroll(p, t)$ individually at some S_{init} in which $tournament(t)$ and $player(p)$ are set to true, generates S_1 and S_2 respectively, which are valid states. However, when we combine the effects of both operations, the resulting state violates the invariant that a player must be enrolled in an existing tournament (line 2, Figure 2).

5.2.2 *Conflict repair*

The conflict repair algorithm (function *repairConflicts*, line 11) takes two conflicting operations and tries to find modifications to the operations and conflict resolution rules that guarantee the invariants are preserved when these operations execute concurrently.

The algorithm starts by creating a pool of predicates to add to the operations to avoid the conflict. To this end, the invariant clauses that might be involved in the conflict are identified, based on the effects of the conflicting operations. The predicates involved in these clauses are the predicates that will be used to try to avoid the conflict (line 13).

The next step is to generate the modified versions of the conflicting operations by using the identified predicates (line 14). The *generate* function (line 21) computes all possible combinations of new effects to add to the operations, by using the powerset of the previously identified predicates (in *invPreds*), filtered so that in each set a predicate has only the value $true$ or $false$ (line 23). The function returns all pairs of operations extending the original operations with the new effects, ignoring, in each new operation, any predicate that is already present in the operation. The modified pairs are ordered by the number of predicates in each operation (line 28) to ensure that the algorithm analyses operations with fewer predicates first.


S_init
Op: rem_tourn(t) Effects: tournament(t) = F
Op: enroll(p,t) Effects: enrolled(p, t) = T
tournament {t}
player {p}
enrolled {}
S1
S2
merge(S1,S2)
S_final
tournament {}
player {p}
enrolled {(p,t)}
Op: rem_tourn(p) Effects: tournaments(t) = F
Op: enroll_t(p,t) Effects: enrolled(p, t) = T tournaments(t) = T
CR: tournament(t) = T (add-wins)
Op: rem_tourn(t) Effects: tournament(t) = F enrolled(*,t) = F
CR: enrolled(p,t) = F (rem-wins)

(a) Referential integrity broken.

(b) Recreates tournament t.

(c) Disenrolls all players from tournament t.

Figure 3: Analysis of conflicts and resolutions of a pair of operations.

For each generated pair of operations, the algorithm checks if the effects of the modified operations are not a subset of any previously found solution, i.e., if there is a solution with fewer additional effects that solves the conflict (line 16). If that is the case, the pair is ignored. Otherwise, the algorithm checks if the new pair is non-conflicting (line 17) – this might require automatically assigning a new conflict resolution policy to a predicate that has no assigned policy yet (if a policy already exists, it is not modified). If the new pair is non-conflicting, it is added to the set of solutions (line 19).

Finally, the set of solutions is presented to the programmer, who must select the most appropriate one for the application (line 20).

5.2.3 Running IPA analysis: an example

To exemplify how the IPA analysis works, we consider three operations: $enroll(p, t)$, $rem\_tourn(t)$, and $rem\_player(p)$.

In the first iteration, the conflict detection procedure finds a pair of operations that conflict. Consider that this returns the pair $\langle enroll(p, t), rem\_tourn(t)\rangle$. In this case, the invariant violated by these operations is $I = enrolled(p, t) \Rightarrow player(p) \wedge tournament(t)$. The pool of predicates for generating new operations is $\{enrolled(p, t), player(p), tournment(t)\}$. The analysis creates new operations by considering the powerset of those effects. While computing the powerset has exponential complexity, the degree is usually small as the predicate pool is restricted to the predicates of the invariant clause impacted by the conflict.

With an initially empty value for the CR set, the following repair actions are possible. The first repair (Figure 3b) consists of extending the $enroll(p, t)$ operation by setting the predicate $tournament(t)$ to $true$ and using the conflict resolution policy (*tournament*, *Add-wins*). When the two operations execute concurrently, an opposing effect occurs for predicate $tournament(t)$. With *Add-wins* for *tournament*, the resulting value for the predicate is $true$, leading to a state that contains tournament t, which is compliant with invariant I. Setting $tournament(t)$ to $true$ in operation $enroll(p, t)$ produces no observable effect because one of the preconditions of the operation is that the value of that predicate is $true$, therefore the effect can be executed preventively. We note that choosing this repair will influence also the result of a concurrent $add\_tourn(t)$ and $rem\_tourn(t)$ – the *Add-wins* policy will guarantee that the tournament exists in that case. Also, this repair would not be possible if the initial conflict resolution policies included (*tournament*, *Rem-wins*).

The second repair (Figure 3c) consists of extending $rem\_tourn(t)$ to force the disenrollment of every player from tournament t and using the conflict resolution policy (*enrolled*, *Rem-wins*). Since it is impossible to infer the identifier of the player that might enroll in t, we use a wildcard for the parameter p, $enrolled(*, t) = false$, as shown in Figure 3c. (We show in Section 6.2 how to implement this predicate efficiently.) This additional effect is also not observable in a sequential execution, and therefore can be applied preventively.

After both solutions are presented to the programmer, he or she selects one of them, and the loop proceeds to find another conflicting pair. The conflict detection step now returns the pair ($enroll(p, t)$, $rem\_player(p)$). The same invariant as before is violated, leading to the same pool of predicates for generating new operations.

Again, independently of the resolution selected for the first conflicting pair, two repairs are possible (as the conflict resolution policies do not restrict them). The first repair consists of extending the $enroll(p, t)$ operation by setting the predicate $player(p)$ to $true$ and using the conflict resolution policy (*player*, *Add-wins*). The second repair consists of extending the $rem\_player(p)$ to force the disenrollment of the player p in every tournament and using the conflict resolution policy (*enrolled*, *Rem-wins*).

Although the most logical solution would appear to be selecting the first or second repair in both cases, it is important to note that any other combination also preserves the invariants. E.g., the programmer might decide that in case of a conflict between $enroll(p, t)$ and $rem\_tourn(t)$, the enroll will win, thus restoring the tournament (first repair), whereas in a conflict between $enroll(p, t)$ and $rem\_player(p)$, the removal of the player will win, leading to the disenrollment of the player in all tournaments (second repair).

A complete analysis is provided in a technical report [15].

5.2.4 Correctness of referential integrity maintenance

We now show that the IPA approach preserves the referential integrity invariant on a generic relational database.

Consider the following definition of referential integrity: $I = \forall x_i \in X, \exists y_i \in Y : x_i.r = v \Rightarrow y_i.s = v$, which says that for every row of table X that has value v for column r, there must be a row in table Y with value v for column s.

Without loss of generality, we consider the case where table Y has a single column and table X has two columns, where the second one refers to a row in table Y. Operation C_x creates a new row (x, y) in a table X, if and only if there is a row in table Y with value y. Operation D_y deletes row (y) in table Y, if and only if there is no row in table X, where the second column has the value y.

Let S_{CD} be the set of admissible states where the preconditions of both C_x and D_y hold. Consider the state s where table X is empty and table Y contains a single row with value (y). State s is an admissible state for both C_x and D_y and thus it is possible to concurrently execute C_x and D_y. However, as $I(C_x(D_y(s)))$ = *false*, the set of operations $\{C_x, D_y\}$ is not *I-Confluent*.

Now consider the modified operation C'_x with the same effects as the original operation, plus an additional effect to touch every row (y) of table Y. Further consider the use of *Add-wins* for table Y, where deletion of a row loses when executed concurrently with an insert or touch. With these modifications in place, the set of operations $O = \{C'_x, D_y\}$ is now *I-Confluent*. To prove that, we are going to show that I holds for every possible execution of operations in O, even if we extend O with other operations that might affect I.

We note that C'_x can only execute initially in states where a row (y) exists in table Y. In these states, the concurrent execution of C'_x and any number of D_y operations always leads to a state where row (y) is present in table Y, since the use of *Add-wins* semantics for table Y guarantees that the touch of (y) in C'_x masks the deletion of (y) in D_y. This guarantees that the invariant I holds for O.

The invariant I continues to hold for the row added by C'_x even if more operations are added to O. Since *Add-wins* semantics are used for table Y, row (y) remains in table Y when C'_x is executed concurrently with any other operation, thus guaranteeing that referential integrity is maintained.

5.3 Numerical invariants

Numerical invariants are more difficult to maintain by modifying operations. For example, consider the invariant in line 4 of Figure 2, which defines a maximum capacity for a tournament. When the current number of enrolled players is $Capacity - 1$, two concurrent enrolls with different players lead to an invariant violation. To avoid this problem, a possible preventive modification is to disenroll some enrolled player when executing an $enroll(p, t)$. (To be correct, different enrolls additionally need to disenroll different players.) Obviously, this is not reasonable in this application.

To circumvent this issue, we provide support for compensations [27, 42, 50]. With compensations, the idea is to check that the precondition holds when executing the operation in the initial replica, and to check that the invariants hold when operations are integrated remotely or when the state is read. Implementations of compensation mechanisms typically require re-executing operations multiple times, or using a leader to order operations [50], to ensure that replicas converge after applying a compensation. We implement compensations without any of these limitations by relying on CRDT convergence rules.

5.3.1 *Extending the analysis*

To generate a compensation, instead of modifying the existing operations, the algorithm flags those operations as conflicting and generates a new operation with the additional effects to be executed when a conflict is detected. Since the new operation may conflict with other pre-existing operations, the analysis must check the execution of this operation against other operations.

The analysis can detect numerical invariant violations that involve comparison operations and arithmetic operators between multiple predicates. For example, an invariant $Stock(x) \geq 0$ can be used to state that the stock of a product must be non-negative and the invariant $CheckAccount(x) + Savings(x) \geq 0$ to specify that the overall balance of a client in a bank must be non-negative.

After detecting a conflict, the analysis suggests effects to repair the conflict. For instance, after identifying a conflict that violates the invariant of line 4, the compensation simply says that the value of that predicate must be decreased, i.e., that the number of players enrolled must decrease. It is up to the programmer to decide how to implement such compensation. In our example, a possible compensation is to disenroll the player and send him a notification.

The logic of compensations is application-dependent. Our tool identifies invariant violations to be fixed using compensations and lets the programmer decide how to fix the violation.

5.3.2 *Extending applications*

Replicas might detect the invariant violation at any time by inspecting the local state and applying a compensation without coordinating with other replicas. The compensation operation is replicated, as any other operation, thus guaranteeing that all replicas converge to the same state. We explain how this works with an example.

```
1  void ensureEnroll ( String p, String t ) {
2      AddWinsSet tournamentIndex = getCRDT(TOUR_IDX, ADD_WINS);
3      AddWinsSet playerIndex = getCRDT(PLR_IDX, ADD_WINS);
4      tournamentIndex.touch( t );
5      playerIndex .touch(p );
6  }
7  void compensateEnrolled( String p, String t ) {
8      AddWinsSet enrolled = getCRDT(ENROLLED_PFX + t, ADD_WINS);
9      RemWinsSet matches = getCRDT(MATCHES_PFX + t, REM_WINS);
10     enrolled .remove(p);
11     matches.rem(new Match(p, "*", t ));
12     matches.touch_rem(new Match(p, "*", t ));
13 }
```

Figure 4: Auxiliary functions for implementing the modified version of the *Tournament* application.

Consider the invariant violation of exceeding the maximum number of allowed players in a tournament. When a violation is detected, some player will have to be disenrolled from the tournament. To fix this violation, the analysis suggests that the programmer decreases the number of players enrolled in the tournament.

Reanalyzing the new specification with this compensation raises new conflicts in the application. The compensation operation conflicts with operation $do\_match()$, as the players in a match must be part of the tournament. Therefore, the compensation must also cancel any match of the player that will be disenrolled. It is up to the programmer to decide which additional effects have to be applied as a consequence of the compensation (e.g., notify the player).

If multiple replicas concurrently detect the invariant violation, they might independently apply the compensation code. As a consequence, different players might be disenrolled in different replicas. In any case, as the effects of the compensation operations are propagated to all replicas, the system converges to a state where invariants are valid. The downside is that we might remove more players from the tournament than necessary. In our application, we use a deterministic rule to decide which player to disenroll to increase the likelihood of disenrolling the same player in all replicas.

5.4 Code modification

After the IPA analysis returns the new specification of the application, the programmer must apply the proposed changes to the code of the application. In our prototype, the programmer is responsible for ensuring that the modified code follows the new specification, although code checking techniques could be used to ensure that the code matches the specification [26].

Figure 4 shows an excerpt of the code necessary to modify the tournament application in our prototype.

The first aspect to consider is how to store information and how to implement the conflict resolution policies. In our prototype, each predicate is represented with one or more set CRDTs: predicates $player(p)$ and $tournament(t)$ are represented by a set, where each element is a member of that set; predicate $enrolled(p, t)$ is represented by a set for each tournament t and each element p is a member of that set. Setting a predicate to $true$ corresponds to adding that element to the set and setting it to $false$ removes it. The set CRDT used for each predicate depends on the conflict resolution policy in the new specification of the application.

In our prototype, using a given conflict resolution policy for a predicate is achieved by using the appropriate CRDT – in function *ensureEnroll* (line 1), the *Add-wins* set CRDT is used for both predicates $player(p)$ and $tournament(t)$.

The second aspect to consider is how to modify the original operations for implementing the new specification, with the additional effects. Instead of adding the code to the original functions, we used auxiliary functions that execute the additional effects and are called by the original operation. For example, function *ensureEnroll* is used to extend the enroll operation to guarantee that player p

and tournament t are not concurrently removed (this corresponds to the first repair for both conflicts in the example of Section 5.2). The code simply uses the touch operation in the CRDTs for both players and tournaments, in combination with the *Add-wins* policy, to guarantee that player p and tournament t are not removed.

When using compensations, it is necessary to write a compensation function that runs if the invariant violation is detected at runtime. Function *compensateEnrolled* (line 7) shows the code for compensating an enroll when it is found that the capacity of the tournament has been exceeded. Besides canceling the enrollment (line 10), it is necessary to cancel any match that involves the player, both the matches that are already known at the replica (line 11) and those that might have been created concurrently (line 12).

5.5 Limitations

No repair found. A limitation of our approach is that the algorithm might not find any valid solution for a conflict between two operations, due to some previous decision by the programmer. Consider, e.g., an application with four predicates, A, B, C and D, connected by the following invariant $A \Rightarrow B \Rightarrow C \Rightarrow D$. For each predicate there are two operations defined, one that makes the predicate true (e.g., A^t) and another that makes it false (e.g., A^f) .

When running the IPA analysis for solving the conflict between operations C^t and D^f, the programmer may decide to repair the conflict by setting both predicates to false. To this end, operation D^f is extended to make $C = \mathit{false}$, and the conflict resolution policy for C is *Rem-wins*.

Next, the conflict detection may identify the conflict between operations A^t and B^f. If the programmer decides to solve the conflict by making both predicates true, operation A^t is extended with $B = \mathit{true}$ and the conflict resolution policy for B is *Add-wins*.

The extended operation A^t now conflicts with operation C^f. However, there is no solution for solving this conflict. First, it is impossible to make predicates A, B and C true, as the conflict resolution for C is *Rem-wins*. Second, it is impossible to extend operation C^f to make the predicate B false, as the conflict resolution for B is *Add-wins*.

In such cases, our tool checks if it would have been possible to solve the conflict by considering only the conflict resolution policies initially established by the programmer. If such a solution exists, it is presented to the programmer. If the programmer wants to use such a solution, because it makes sense for the semantics of the application being developed, he or she must run the analysis again, and use its output to make different decisions on the alternative repairs that are proposed. Otherwise, the pair is flagged as unsolvable and the algorithm continues, ignoring that pair in subsequent iterations. In that case, the execution of those operations must be controlled using an alternative mechanism [14, 37].

Supporting multiple applications and application evolution. Our approach can support multiple applications accessing the same database if the programmer provides a complete specification for all applications that use the database.

For a running system, one can consider several scenarios of evolution: add new operations to an application; add new invariants to the database; add a new application that accesses the same database. In all cases, it is necessary to execute the IPA analysis again. If the operations and conflict resolution rules used in the original system remain unchanged, it is possible to evolve the system without having to stop it. Otherwise, it is necessary to stop the system.

A possible approach to tackle the system evolution more easily is to consider that, for every predicate defined in the system, there exists an operation to make the predicate true and an operation to make it false, even if those operations do not exist in the current application. Considering these operations, the IPA analysis will identify conflicts between the operations that exist in the application and these additional operations, and proposes alternative repairs to the programmer. With this approach, when evolving the system, if the database does not change and no additional invariant is added, as it is often the case when supporting a new application that uses the same database, the original operations will remain unchanged (as they already prevent any conflict that may occur). This approach has two main disadvantages. First, the operation may include additional effects that are of no use in the current application (as the conflicts they are solving do not occur). Second, the programmer may decide to use a repair approach that is not appropriate for the new operations, making this effort useless.

Specification effort. The effort of writing specifications is arguably comparable to the effort of writing the code itself [43]. We do not address this problem in this work. Previous research attempted to address this problem, proposing automatic feature extraction and code synthesis to aid the programmer in writing correct applications [44, 36, 24, 25, 7]. Our approach could benefit from these complementary research efforts, not only for extracting the specification of the application from the code but also for making sure that the code of the modified version of the application matches the specification output by the IPA analysis.

6. IMPLEMENTATION

This section briefly describes the IPA prototype.

6.1 IPA tool and database support

The IPA tool assists programmers to write invariant preserving applications. The tool receives as input an annotated Java interface with the operations and the invariants, as in the example of Figure 2, and an initial set of convergence rules. The tool runs the static analysis algorithm and outputs the modified specification of operations and auxiliary compensations, the conflict resolution rules for each predicate, and the set of unresolved conflicting pairs.

The tool uses the Z3 SMT solver [21] to identify conflicting operations and propose modifications to operations. Boolean satisfiability is an NP-Complete problem, but modern SMT solvers can handle many instances of this problem efficiently, as shown by our performance evaluation. The algorithms and the tool are implemented using standard Java and the Java bindings for Z3.

Our prototype was implemented in Java, using Switfcloud [57, 3] as the underlying storage system. SwiftCloud provides highly available transactions, causal consistency and per-object conflict resolution based on CRDTs, allowing to easily add new data types.

We implemented several applications for the evaluation, derived from the specifications generated by our analysis. We use set CRDTs for storing the data represented by the predicates.

6.2 New CRDT designs

We now discuss the CRDTs used for supporting the resolutions proposed by IPA. A detailed specification of the data types is available in a separate document [48].

6.2.1 CRDTs with touch operations

When a modification requires that some predicate value is set to true, we need to ensure that the element that we are *restoring* is equal to the one that was observed.

To this end, we extended the *Add-wins* set CRDT with a *touch* operation. This operation simply updates the meta-data for the element in the set, with a new timestamp as if the object was created at that moment. If a concurrent remove of the same element is executed, the *Add-wins* policy ensures that the element will survive.

Table 1: Common classes of invariants in applications.

Inv. Type	*I*-Conf.	IPA	TPC	Tour	Ticket	Twitter
Sequential id.	No	No	Yes	—	—	—
Unique id.	Yes	Yes	Yes	Yes	Yes	Yes
Numerical inv.	No	Comp.	Yes	—	—	—
Aggreg. const.	No	Comp.	—	—	—	—
Aggreg. incl.	Yes	Yes	—	Yes	—	—
Ref. integrity	No	Yes	Yes	Yes	—	Yes
Disjunctions	No	Yes	—	Yes	—	—

Furthermore, all information associated with the element is preserved – for example, for a tournament, the element includes not only a simple identifier but a record with multiple fields.

6.2.2 Remove with Wildcards

Some repairs require preventing the addition of any element that matches some condition – e.g., in the example of Section 5.2, there is a repair requiring to make $enrolled(*, t) = false$, for any player and a given tournament. To implement this effect using a standard *Rem-wins* set CRDT, it would be necessary to execute a remove for all elements that could be concurrently added.

Doing this would be impractical because the set of possible elements is large. Instead, we created CRDTs that support *wildcard values* for the remove operation. For instance, the wildcard "*" represents all possible elements in the domain.

6.2.3 Compensation CRDTs

For some invariants, it is possible to encapsulate the logic for detecting conflicts and applying the compensations automatically. Consider the invariant $\#enrolled(p, t) \leq Capacity$. We use a set to store the information about players enrolled in a tournament. To ensure that the application is always consistent, whenever the application accesses the set, the object automatically verifies if its state is consistent, and applies the compensation if necessary.

We implemented *Limited Size Set CRDTs* that allow the programmer to define the constraint that must be maintained, and the compensation to execute when the constraint is violated. Whenever the object is read, the code is executed automatically, ensuring that any observed state is consistent. The effects of the compensation, in case it is executed, are committed in the enclosing transaction.

To select the elements to be removed in the compensation it is possible to leverage the history information of the set to remove the last elements added to the set. This does not prevent more elements than necessary from being removed, because the state of replicas might diverge, but it reduces the chance of that happening.

7. EVALUATION

Our evaluation intends to answer the following questions:

(i) What classes of invariants and applications can be handled by our approach?

(ii) What is the scalability of the static analysis process?

(iii) How does the performance of modified applications compare to other solutions?

7.1 Invariants covered by IPA

This section surveys the invariants covered by our approach.

7.1.1 Classes of invariants

Prior work has analyzed invariants used in real applications [10, 9, 37]. Table 1 summarizes whether these invariants are *I-Confluent* [9] or can be made *I-Confluent* by using *IPA*.

Sequential identifiers: Sequential identifiers are useful to enforce a total order of elements. In general, generating these identifiers requires coordination to avoid collisions. No solution based on weak consistency can maintain this invariant. However, it has been shown that, in most cases, applications could easily replace the use of sequential identifiers by unique identifiers [8, 51].

Unique identifiers: Unique identifiers can be preserved without coordination, and as such are *I-Confluent*. Unique identifiers do not provide a sequential order, but can still provide a total order of identifiers compatible with the happens-before relation.

Numerical invariants: Numerical invariants assert conditions involving numerical predicates (e.g., $p(x) < k$). In general, preserving these invariants requires coordination. However, as it has been shown, it is possible to enforce some constraints on top of weak consistency by relying on escrow techniques [16, 32, 41]. IPA allows to maintain these invariants using compensations as long as the application is compatible with using this approach. In TPC-C/W we can use compensations to replenish the stock, for instance.

Aggregation constraint: Imposing a bound on the size of a collection, e.g., limiting the players enrolled in a tournament, can be addressed using a numerical invariant over a predicate that represents the size of the collection, thus sharing its properties.

Aggregation inclusion: Ensuring that an element is eventually added or removed from a collection is *I-Confluent*, provided no dependencies to other objects exist. If that is not the case, referential integrity might be required.

Referential integrity: Preserving relations and dependencies among objects, such as foreign keys in relational databases and references to keys in key-value stores, is not *I-Confluent*. IPA fully supports this invariant, as exemplified throughout the paper.

Disjunctions: Applications often specify that one of several conditions must be met by using a disjunction. IPA can address this type of invariant by extending an operation to ensure that the disjunction is always true. This is an extension of the mechanism for supporting referential integrity, as in this case there might be several alternative conditions that restore the validity of the invariant.

7.1.2 Invariants in applications

We now analyze how IPA can address the invariants of some selected applications (summarized in Table 1). These application are representative of general OLTP workloads.

***Tournament*:** This application manages the information for an on-line game and showcases most of the invariants that our solution can address. It is based on an application used in prior work [29, 14] with a few new constraints. For this application, IPA is capable of proposing multiple alternative resolutions that either reconstruct broken dependencies, or clear them, to avoid inconsistencies due to concurrent executions, as discussed throughout this paper.

***Twitter*:** We implemented a simple Twitter clone that relies heavily on referential integrity to implement user timelines and maintain subscription information. When some user tweets, we write those tweets to the timelines of the followers. This leads to consistency issues when tweets and users are removed concurrently with tweeting. We implemented two versions, using the *Add-wins* and the *Rem-wins* policy to solve conflicts, respectively. For example, if a user tweets and his or her account is concurrently removed, for the *Add-wins* version the user and tweets are restored, whereas for *Rem-wins* the user and tweets are removed. Other conflicts are solved similarly.

***Ticket*:** This application, based on FusionTicket [1, 32, 56], manages ticket reservations. The main invariant is that tickets for events cannot be oversold. We use the *Limited Size Set CRDT* to cancel a ticket sale and reimburse the customer when tickets are oversold. The transfer of money to the account of the customer crosses the boundaries of the system and therefore must use a different mechanism (e.g., a message queue).

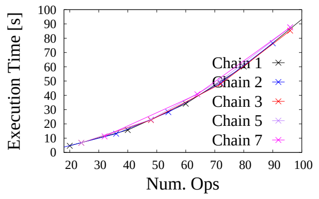


Figure 5: Execution time of analysis with no conflict.

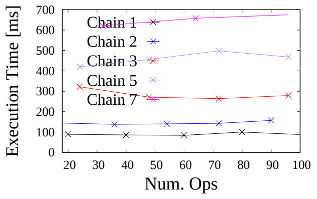


Figure 6: Time to generate repairs.

***TPC-W* and *TPC-C*:** These standard database benchmarks overlook some aspects of real-world applications, such as having operations to manage product listings. In our specification, we extended these applications to include such operations, which introduced referential integrity constraints. To implement the inventory size threshold we used compensations to increase the stock (in accordance with the specification). An alternative would be to cancel the oversold purchases, as in the previous example.

7.1.3 Summary

The types of invariants we support (Table 1) are common in many applications, as previously shown [10]. The examples we discussed show that our language is expressive enough to address complex applications, including typical relational database applications.

7.2 Off-line analysis performance

We evaluate the scalability of the proposed approach, by analyzing the execution time of the IPA analysis as the size and complexity of the application increases using a micro-benchmark.

To vary the size of the application, we vary the number of predicates. For each predicate, there are two operations, one that makes the predicate true and another that makes the predicate false. Thus, for 10 predicates, we have 20 operations. For all predicates, we use the *Add-wins* conflict resolution policy.

All predicates belong to some invariant clause. To vary the complexity, we vary the number of predicates that are connected by invariants. For a chain size of one, the invariants defined connect only pairs of predicates: $P_1 \Rightarrow P_2$, $P_3 \Rightarrow P_4$, $P_5 \Rightarrow P_6$, etc. For a chain size of two, the invariants connect groups of three predicates: $P_1 \Rightarrow P_2$ and $P_2 \Rightarrow P_3$, $P_4 \Rightarrow P_5$ and $P_5 \Rightarrow P_6$, etc. Increasing the size of the chain, increases both the number of clauses in the invariant and the average number of effects in each operation for a specification without conflicts – e.g., with a chain of size one, the operation that makes P_1 true only needs to make P_2 also true, while with a chain of size two it also needs to make P_3 true.

The results were obtained in a laptop, running MacOS 10.13.6, with an Intel i7 2.8 GHz Quad-Core processor and 16GB of RAM.

7.2.1 Conflict detection

We start by evaluating the time to detect a conflict. The worst case scenario is when the analysis finds the conflict in the last pair that is checked. To approximate this, we run our tool with a correct specification, so that the algorithm analyzes all pairs of operations. Figure 5 shows the execution time (in seconds) for different configurations of the benchmark, with different lines corresponding to increasing the number of operations in the chain.

We first observe that, as expected, the execution time of the algorithm grows quadratically, due to testing all pairs of operations. Second, we observe that the cost of testing all pairs dominates the cost of the algorithm, since changing the size of the chain does not impact the results significantly. The results show that the overall execution time is reasonable for an offline process, as testing all combinations of pairs of 100 operations takes less than 100 seconds.

7.2.2 Conflict repair

Our second experiment evaluates the time to repair conflicts. To this end, from a specification that is invariant-preserving, we have removed the additional effects that avoid conflicts in a single operation, selecting one of the operations with more additional effects (e.g., in the chain of size two, from the operation that makes P_1 true, we removed the changes to P_2 and P_3). Figure 6 presents the sum of the time spent in proposing repairs until a correct solution is generated (this excludes the time to detect conflicts). As expected, when the chain is longer, our tool takes more time to find a correct solution, as it repairs the conflicts one at a time. Still, in our configurations, this time is low.

These results show that the running time of our tool is dominated by the time to detect conflicts and that this time is reasonable for an offline process, making our approach practical.

7.3 Runtime performance

In this section, we compare the performance of applications modified using our approach against alternative solutions.

7.3.1 System configurations

Our evaluation was performed in a geo-replicated setting on Amazon EC2. The database deployment consists of three servers running in three geographical regions (US-WEST, US-EAST and EU-IE). The table below shows the latency between each region.

RTT (ms)	US-East	US-West
US-West	81	–
EU-IE	93	161

The application server is co-located with the storage system of each region. We use SwiftCloud to implement all different approaches that we evaluate. Clients are installed in other machines in the same availability zones as the corresponding closest servers.

Performance of applications is compared with the following alternative approaches:

Causal Consistency (*Causal*): This configuration uses the original applications running with causal consistency, which does not maintain invariants for conflicting operations.

Strong Consistency (*Strong*): All updates are forwarded to the US-EAST replica to enforce serialization. This minimizes the average latency for updates.

Invariant violation avoidance (*Indigo* [14]): Applications modified to use the efficient coordination mechanisms proposed in Indigo [2] to prevent invariant violations. Conflicting operations need to acquire reservations to safely execute operations. Each set of conflicting operations coordinated using different reservations, and each reservation can be shared by multiple replicas, allowing a high level of concurrency.

Invariant Preserving Applications (IPA): Applications are modified using our IPA approach, which maintains invariants without coordination, on top of *Causal*.

7.3.2 Throughput and latency

We evaluate the scalability of each configuration by measuring the latency and throughput with different loads on the system, using the *Tournament* application. The workload comprises 35% of write operations, and the initial database contains 1000 players and 100 tournaments. All operations are conflicting in the original application. In the IPA modified version, all operations are *I-Confluent* using a mix of conflict resolution policies. In Indigo, every pair of operations is protected by a different reservation.

To test the scalability of the system, we increase the number of clients contacting each server by running extra client threads until peak throughput is achieved.

Figure 7 shows that *Strong* presents the highest average latency, which is a consequence of having $\frac{2}{3}$ of operations being forwarded to a remote server. Even if it would be possible to improve the scalability of our implementation, this figure highlights that even when contention is low, the average latency is already much higher than with weak consistency approaches. *Causal* shows the best scalability with the lowest latency. Our approach, IPA, performs slightly worse than *Causal*, as additional updates need to be executed to preserve application invariants. In the *Causal* version, concurrent updates may lead to invariant violation.

When compared to *Indigo*, our approach performs slightly better. The reason for this is that updates in *Indigo* need to acquire reservations for coordinating the execution of concurrent updates, which is more costly than the additional updates in the IPA version. The advantage is small because reservations are exchanged among replicas infrequently after they are acquired. Additionally, as many reservations can be shared, they allow a high degree of concurrency.

Figure 8 presents the latency for the write operations and highlights more clearly the differences between the configurations. We omit the *Strong* column. The average latency of operations in *Indigo* is higher than the latency for IPA or *Causal* and also exhibits a higher standard deviation. Both are explained by the occasional need for *Indigo* replicas to trade reservations. Compared to *Causal*, the latency of write operations is only slightly higher in IPA, which is due to the extra updates executed. We further study the overhead associated with executing extra effects in Section 7.3.5.

7.3.3 Comparing different conflict resolution schemes

We implemented *Twitter* using *Add-wins* and *Rem-wins* strategies to compare the costs of each approach. The initial database contains 1000 users, and 96% of the operations are writes.

Figure 9 shows the latency of each operation for the different version. The *Add-wins* version has a higher latency for operations that create tweets (tweet and retweet). This overhead is due to the additional effect of *touching* the user, to ensure that when a user tweets, or retweets, he or she will not be removed concurrently.

The *Rem-wins* version has a higher latency for remove operations, due to the additional effects. The delete tweet operation needs to remove the tweet from all timelines that have the tweet. The remove user has a small overhead, as it only has to issue a remove with a wildcard in the followers and to set a compensation for removing tweets in the user object – when user u_1 reads a timeline, the application checks, for each tweet, that its author, u_2, was removed, triggering a compensation to remove the tweets from the timeline of u_1 in this case. This also leads to a slight overhead in the read timeline operation.

7.3.4 Scalability of compensations

We evaluate the scalability of the compensation mechanism implemented in the Limited Size Set CRDT with the *Ticket* application, by increasing contention. The initial database has 500 flights and 10000 customers, and all operations in the workload read and update the database, with reads triggering the execution of compensations when a flight is overbooked.

Figure 10 presents the performance with an increasing load. The red dots in the figure indicate the average number of invariant violations that were observed at that throughput, when using *Causal*. This confirms the intuition that as contention rises, the divergence window grows larger, increasing the chance for invariant violation. In *Causal*, this exposes the application to consistency anomalies, while in IPA executing compensations preserves invariants at all times. As expected, compensations incur on some overhead, but still provide latency comparable to *Causal*.

7.3.5 Microbenchmarks

IPA avoids invariant violations by executing extra updates in one or more objects. In this section, we evaluate the overhead of adding additional effects to operations. We analyze the impact of executing increasingly more updates in comparison to the costs of executing the original operation in strong consistency and Indigo. These microbenchmark use a Set CRDT to store information.

Operations on a single object: We measure the speedup of an application modified with IPA versus the original operation running on *Strong*. Figure 11 (top) shows that the original operation is about $28\times$ faster in IPA than in *Strong*. Adding more updates to this operation makes the speedup decrease. When we execute 2048 updates to a single object, the average latency is still about 40ms.

Operations over multiple objects: Executing updates on a single object imposes a low overhead on the system, because the object is read and written to storage only once and subsequent updates only impose processing costs. Now we evaluate the overhead when the additional effects of modified operations update multiple objects.

The original application reads a varying number of objects to check some condition and then executes a single write operation to an object. The modified application checks the same condition, but executes a write for each object. Figure 11 (bottom) shows performance dropping faster than when executing updates over single objects. At 64 objects, it starts to pay off to switch to *Strong*.

In practice, in the evaluated applications, we require only a few extra updates per object over a small number of objects. In *Twitter*, which needs to execute more writes due to our implementation of the timeline, we were able to execute them lazily via compensations.

Comparison with *Indigo*: In Indigo [14], operations might execute locally if the replica holds some specific reservation. Multiple operations might be able to execute concurrently at different replicas if all of them can share the same reservation. If a replica requires some reservation that is being used exclusively, the replica must request remote replicas to release the reservation, before acquiring it. This approach only avoids coordination when a replica holds the necessary reservations to execute the operation. Thus, the latency of an application depends on the contention for obtaining reservations.

In this experiment, we evaluate the impact of varying the percentage of operations that compete to acquire opposing reservations. We compare the performance of this solution against executing the same operation in IPA. Figure 12 shows that the performance of IPA is equivalent to *Indigo* with no contention for reservations, and that the latency of *Indigo* rises steadily as contention increases.

Despite the overhead for executing the additional effects, IPA provides a predictable latency of operations, which is not the case for *Indigo*, whose operations latency depend on the current distribution of reservations. Furthermore, our approach is fault-tolerant as a client can execute operations as long as it can access a single server. In *Indigo*, if a server that holds the reservation needed to execute some operation becomes unavailable, the operation cannot execute.

8. RELATED WORK

Achieving low latency, high availability and data consistency in distributed systems is difficult, as postulated in the CAP theorem [28]. In recent years, researchers and practitioners have studied the trade-offs in distributed systems to provide the best consistency guarantees for different types of applications [20, 22, 39, 8, 37, 14].

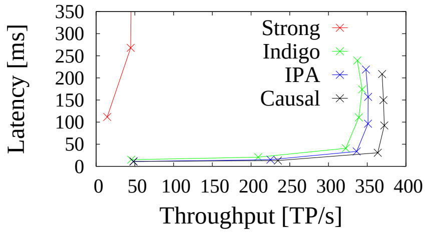


Figure 7: Peak throughput for *Tournament*.

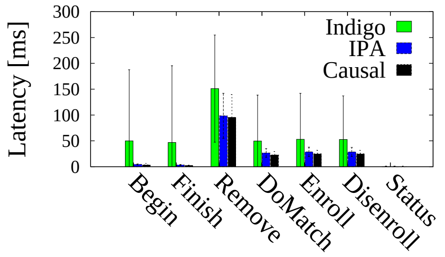


Figure 8: Latency of individual operations in *Tournament*.

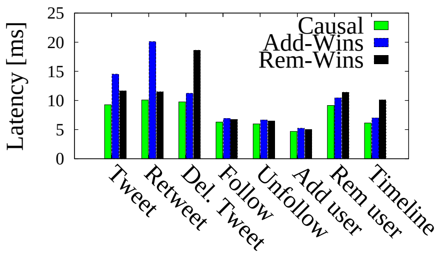


Figure 9: Latency of individual operations in *Twitter*.

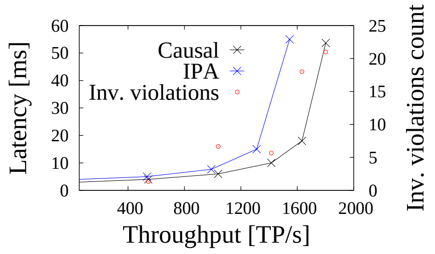


Figure 10: Peak throughput for *Ticket* benchmark. Red dots indicate number of invariant violations observed during runtime.

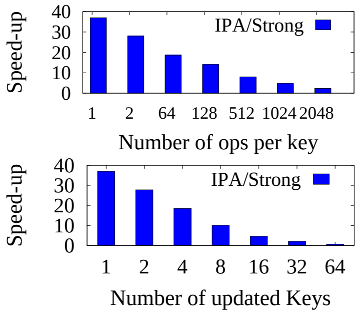


Figure 11: Speedup of executing multiple writes in IPA versus *Strong*.

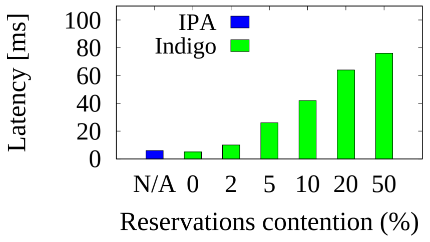


Figure 12: Latency of operations with varying percentage of reservation types in comparison to no reservations.

Systems that ensure strong consistency [20, 58, 23, 33] require coordination across replicas, which is expensive in geo-replicated scenarios. In Megastore [13], data is partitioned at a fine granularity to achieve low latency, while MDCC [33] exploits commutativity and protocol optimizations to improve performance. Spanner [20] and Farm [23] harness custom hardware to improve performance.

Systems that use weak consistency are widely deployed in the real-world [47, 34], but can be difficult to use [10]. Many systems provide causal consistency coupled with object convergence and transactions [39, 57, 40, 8], which all can be implemented efficiently without hindering the availability of the system.

Convergent data types [45] provide automatic replica convergence, which lessens the programming effort in these systems. However, data type convergence alone cannot prevent invariant violations involving multiple objects. To mitigate the problem, RedBlue [37] and Walter [49] provide support for executing operations under weak or strong consistency to allow fast operations when invariants are not at risk, and consistent operations otherwise. Sieve [36] and Blazes [7] address the problem of automating the use of the most appropriate consistency alternative, while Indigo [14], Olisipo [38], Lucy [54], and the Homeostasis protocol [44] try to minimize the use of the strong consistency path. Despite improving the latency of operations in the general case, systems that depend on coordination to execute some operations may still become unavailable and exhibit high latency. IPA completely avoids the drawbacks of coordination, while being able to preserve a wide range of invariants.

Helland and Campbell have suggested that applications should handle invariant violations as part of the application logic, as an alternative to executing operations under strong consistency to prevent violations [31]. The idea of compensations [27] is to execute operations optimistically and explicitly rollback the effects when conflicts are detected. A few systems have explored this model. In PLANET [42], transactions execute speculatively, allowing the system to provide the control back to the client before the transaction commit confirmation arrives. In Bayou [50], transactions commit locally and remain in a tentative state, until all replicas agree on the ordering of operations. Existing systems that use compensations still use some form of coordination to commit transactions. Our approach departs from this model by modifying the operations to ensure they can always commit locally, while preserving invariants. We show that our approach does not modify the semantics of operations when no conflicting concurrent operations execute.

Recent papers focused on proving correctness of distributed systems [30, 55]. These proposals complement ours, as they focus on attesting if implementations conform to some specification, whereas we aim to provide a methodology for implementing correct applications on top of the assumptions of our chosen consistency model.

9. CONCLUSION

In this paper, we proposed a novel approach for supporting correct and highly available applications on top of weak consistency. By extending operations with additional effects, we can ensure invariant preservation at all times with sensible semantics. Our IPA analysis and tool assist the programmer via static analysis to identify which operations might lead to an invariant violation, when executed concurrently, and by suggesting modifications to the operations.

Our experimental evaluation shows that the static analysis can handle large applications in reasonable time for an offline process, and that the modified applications have similar performance to their unmodified counterparts that do not preserve invariants.

Acknowledgments

We thank the anonymous reviewers for their comments that helped improving the paper. This work was partially supported by EU H2020 LightKone project (732505), and FCT/MCTES grants SFRH/BD/87540/2012, UID/CEC/04516/2013, UID/CEC/50021/2013, Lisboa-01-0145-FEDER-032662 /PTDC/CCI-INF/32662/2017, and PTDC/CCI-INF/32081/2017. Computing resources were provided by an Amazon Web Services (AWS) in Education Research Grant.

10. REFERENCES

[1] Fusion Ticket. `http://www.fusionticket.org/`. Accessed May-2018.

[2] Indigo source code. `https://github.com/SyncFree/Indigo`. Accessed May-2018.

[3] SwiftCloud source code. `https://github.com/SyncFree/SwiftCloud`. Accessed May-2018.

[4] Akka. Distributed Data. `https://doc.akka.io/docs/akka/current/distributed-data.html`. Accessed May-2018.

[5] D. D. Akkoorath, A. Z. Tomsic, M. Bravo, Z. Li, T. Crain, A. Bieniusa, N. Preguiça, and M. Shapiro. Cure: Strong semantics meets high availability and low latency. In *Proceedings of the 36th IEEE International Conference on Distributed Computing Systems (ICDCS 2016)*, pages 405–414, Nara, Japan, June 2016.

[6] S. Almeida, J. Leitão, and L. Rodrigues. ChainReaction: A Causal+ Consistent Datastore Based on Chain Replication. In *Proceedings 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 85–98, Prague, Czech Republic, 2013. ACM.

[7] P. Alvaro, N. Conway, J. M. Hellerstein, and D. Maier. Blazes: Coordination analysis for distributed programs. In *Proceedings of the IEEE 30th International Conference on Data Engineering March 31 - April 4, 2014*, pages 52–63, Chicago, Illinois, USA, Apr. 2014.

[8] P. Bailis, A. Davidson, A. Fekete, A. Ghodsi, J. M. Hellerstein, and I. Stoica. Highly available transactions: Virtues and limitations. *PVLDB*, 7(3):181–192, 2013.

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

[10] P. Bailis, A. Fekete, M. J. Franklin, A. Ghodsi, J. M. Hellerstein, and I. Stoica. Feral Concurrency Control: An Empirical Investigation of Modern Application Integrity. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 1327–1342, Melbourne, Victoria, Australia, 2015. ACM.

[11] P. Bailis, A. Fekete, J. M. Hellerstein, A. Ghodsi, and I. Stoica. Scalable Atomic Visibility with RAMP Transactions. In *Proceedings 2014 ACM SIGMOD Conference Conference on Management of Data*, SIGMOD '14, pages 27–38, New York, NY, USA, 2014. ACM.

[12] P. Bailis, A. Ghodsi, J. M. Hellerstein, and I. Stoica. Bolt-on Causal Consistency. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 761–772, New York, New York, USA, 2013. ACM.

[13] J. Baker, C. Bond, J. C. Corbett, J. Furman, A. Khorlin, J. Larson, J.-M. Leon, Y. Li, A. Lloyd, and V. Yushprakh. Megastore: Providing Scalable, Highly Available Storage for Interactive Services. In *Proceedings of the Conference on Innovative Data system Research (CIDR)*, pages 223–234, 2011.

[14] V. Balegas, S. Duarte, C. Ferreira, R. Rodrigues, N. Preguiça, M. Najafzadeh, and M. Shapiro. Putting Consistency Back into Eventual Consistency. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, pages 6:1–6:16, Bordeaux, France, 2015. ACM.

[15] V. Balegas, N. Preguiça, S. Duarte, C. Ferreira, and R. Rodrigues. IPA: Invariant-preserving Applications for Weakly-consistent Replicated Databases. *CoRR*, abs/1802.08474, 2018.

[16] V. Balegas, D. Serra, S. Duarte, C. Ferreira, M. Shapiro, R. Rodrigues, and N. Preguiça. Extending Eventually Consistent Cloud Databases for Enforcing Numeric Invariants. In *Proceedings of the 34th IEEE Symposium on Reliable Distributed Systems (SRDS)*, pages 31–36, Montreal, Canada, Sept 2015.

[17] Basho. Riak. `http://basho.com/`, 2017. Accessed May-2018.

[18] C. Biyikoglu. Under the Hood: Redis CRDTs. `https://goo.gl/tGqU7h`. Accessed May-2018.

[19] N. Conway, W. R. Marczak, P. Alvaro, J. M. Hellerstein, and D. Maier. Logic and Lattices for Distributed Programming. In *Proceedings of the 3rd ACM Symposium on Cloud Computing*, SoCC '12, pages 1:1–1:14, San Jose, California, 2012. ACM.

[20] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford. Spanner: Google's Globally-distributed Database. In *Proceedings 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 251–264, Hollywood, USA, 2012. USENIX Association.

[21] L. De Moura and N. Bjørner. Z3: An Efficient SMT Solver. In *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, pages 337–340. Springer-Verlag, Budapest, Hungary, 2008.

[22] 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 the 21st ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 205–220, Stevenson, Washington, USA, 2007. ACM.

[23] A. Dragojević, D. Narayanan, E. B. Nightingale, M. Renzelmann, A. Shamis, A. Badam, and M. Castro. No Compromises: Distributed Transactions with Consistency, Availability, and Performance. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 54–70, Monterey, California, 2015. ACM.

[24] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The Daikon System for Dynamic Detection of Likely Invariants. *Science of Computer Programming*, 69(1-3):35–45, Dec. 2007.

[25] C. Flanagan and K. R. M. Leino. Houdini, an Annotation Assistant for ESC/Java. In *Proceedings of the International Symposium of Formal Methods Europe on Formal Methods for Increasing Software Productivity*, FME '01, pages 500–517, Berlin, Germany, 2001. Springer-Verlag.

[26] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended Static Checking for Java. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, PLDI '02, pages 234–245, Berlin, Germany, 2002. ACM.

[27] H. Garcia-Molina and K. Salem. Sagas. In *Proceedings of the 1987 ACM SIGMOD International Conference on Management of Data*, SIGMOD '87, pages 249–259, San Francisco, USA, 1987. ACM.

[28] 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.

[29] A. Gotsman, H. Yang, C. Ferreira, M. Najafzadeh, and M. Shapiro. 'Cause I'm Strong Enough: Reasoning About Consistency Choices in Distributed Systems. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, pages 371–384, St. Petersburg, USA, 2016. ACM.

[30] C. Hawblitzel, J. Howell, M. Kapritsos, J. R. Lorch, B. Parno, M. L. Roberts, S. Setty, and B. Zill. IronFleet: Proving Practical Distributed Systems Correct. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 1–17, Monterey, USA, 2015. ACM.

[31] P. Helland and D. Campbell. Building on quicksand. In *Online Proceedings of CIDR 2009 Fourth Biennial Conference on Innovative Data Systems Research*, Asilomar, USA, Jan. 2009.

[32] B. Holt, J. Bornholt, I. Zhang, D. Ports, M. Oskin, and L. Ceze. Disciplined Inconsistency with Consistency Types. In *Proceedings of the 7th ACM Symposium on Cloud Computing*, SoCC '16, pages 279–293, Santa Clara, USA, 2016. ACM.

[33] T. Kraska, G. Pang, M. J. Franklin, S. Madden, and A. Fekete. MDCC: Multi-data Center Consistency. In *Proceedings 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 113–126, Prague, Czech Republic, 2013. ACM.

[34] A. Lakshman and P. Malik. Cassandra: A Decentralized Structured Storage System. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, Apr. 2010.

[35] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558–565, July 1978.

[36] C. Li, J. Leitão, A. Clement, N. Preguiça, R. Rodrigues, and V. Vafeiadis. Automating the Choice of Consistency Levels in Replicated Systems. In *Proceedings of the 2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 281–292, Philadelphia, PA, June 2014. USENIX Association.

[37] C. Li, D. Porto, A. Clement, J. Gehrke, N. Preguiça, and R. Rodrigues. Making Geo-replicated Systems Fast As Possible, Consistent when Necessary. In *Proceedings 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 265–278, Hollywood, USA, 2012. USENIX Association.

[38] C. Li, N. Preguiça, and R. Rodrigues. Fine-grained consistency for geo-replicated systems. In *Proceedings of the 2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 359–372, Boston, MA, 2018. USENIX Association.

[39] 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 23rd ACM Symposium on Operating Systems Principles*, SOSP '11, pages 401–416, Cascais, Portugal, 2011. ACM.

[40] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Stronger Semantics for Low-latency Geo-replicated Storage. In *Proceedings 10th USENIX Conference on Networked Systems Design and Implementation*, NSDI'13, pages 313–328, Lombard, USA, 2013. USENIX Association.

[41] P. E. O'Neil. The Escrow Transactional Method. *ACM Trans. Database Syst.*, 11(4):405–430, Dec. 1986.

[42] G. Pang, T. Kraska, M. J. Franklin, and A. Fekete. PLANET: Making Progress with Commit Processing in Unpredictable Environments. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 3–14, Snowbird, Utah, USA, 2014. ACM.

[43] D. L. Parnas. Precise Documentation: The Key to Better Software. In S. Nanz, editor, *The Future of Software Engineering*, pages 125–148. Springer, 2010.

[44] S. Roy, L. Kot, G. Bender, B. Ding, H. Hojjat, C. Koch, N. Foster, and J. Gehrke. The Homeostasis Protocol: Avoiding Transaction Coordination Through Program Analysis. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 1311–1326, Melbourne, Victoria, Australia, 2015. ACM.

[45] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. A comprehensive Study of Convergent and Commutative Replicated Data Types. Research Report RR-7506, INRIA, Jan. 2011.

[46] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. Conflict-free Replicated Data Types. In *Proceedings of the 13th Conference Conference on Stabilization, Safety, and Security of Distributed Systems*, SSS'11, pages 386–400, Grenoble, France, 2011. Springer-Verlag.

[47] 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, Scottsdale, Arizona, USA, 2012. ACM.

[48] V. B. Sousa. *Invariant Preservation in Geo-replicated Data Stores*. PhD thesis, Faculdade de Ciências e Tecnologia, Universidade NOVA de Lisboa, 12 2017.

[49] Y. Sovran, R. Power, M. K. Aguilera, and J. Li. Transactional Storage for Geo-replicated Systems. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*, SOSP '11, pages 385–400, Cascais, Portugal, 2011. ACM.

[50] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, SOSP '95, pages 172–182, Copper Mountain, Colorado, USA, 1995. ACM.

[51] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden. Speedy Transactions in Multicore In-memory Databases. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles*, SOSP '13, pages 18–32, Farminton, Pennsylvania, USA, 2013. ACM.

[52] W. Vogels. Eventually consistent. *Communications of the ACM*, 52(1):40–44, Jan. 2009.

[53] T. Warszawski and P. Bailis. ACIDRain: Concurrency-Related Attacks on Database-Backed Web Applications. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, pages 5–20, Chicago, Illinois, USA, 2017. ACM.

[54] M. Whittaker and J. M. Hellerstein. Interactive checks for coordination avoidance. *PVLDB*, 2(1):14–27, 2018.

[55] J. R. Wilcox, D. Woos, P. Panchekha, Z. Tatlock, X. Wang, M. D. Ernst, and T. Anderson. Verdi: A Framework for Implementing and Formally Verifying Distributed Systems. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, pages 357–368, Portland, OR, USA, 2015. ACM.

[56] C. Xie, C. Su, M. Kapritsos, Y. Wang, N. Yaghmazadeh, L. Alvisi, and P. Mahajan. Salt: Combining acid and base in a distributed database. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 495–509, Broomfield, CO,

USA, 2014. USENIX Association.

[57] M. Zawirski, N. Preguiça, S. Duarte, A. Bieniusa, V. Balegas, and M. Shapiro. Write Fast, Read in the Past: Causal Consistency for Client-Side Applications. In *Proceedings of the 16th Annual Middleware Conference*, Middleware '15, pages 75–87, Vancouver, BC, Canada, 2015. ACM.

[58] Y. Zhang, R. Power, S. Zhou, Y. Sovran, M. K. Aguilera, and J. Li. Transaction Chains: Achieving Serializability with Low Latency in Geo-distributed Storage Systems. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles*, SOSP '13, pages 276–291, Farminton, Pennsylvania, USA, 2013. ACM.