AutoGR: Automated Geo-Replication with Fast System Performance and Preserved Application Semantics

Jiawei Wang, Cheng Li, Kai Ma, Jingze Huo, Feng Yan, Xinyu Feng, Yinlong Xu

ABSTRACT
Geo-replication is essential for providing low latency response and quality Internet services. However, designing fast and correct geo-replicated services is challenging due to the complex trade-off between performance and consistency semantics in optimizing the expensive cross-site coordination. State-of-the-art solutions rely on programmers to derive sufficient application-specific invariants and code specifications, which is both time-consuming and error-prone. In this paper, we propose an end-to-end geo-replication deployment framework AutoGR (AUTOmated Geo-Replication) to free programmers from such label-intensive tasks. AutoGR enables the geo-replication features for non-replicated, serializable applications in an automated way with optimized performance and correct application semantics. Driven by a novel static analyzer Rigi, AutoGR can extract application invariants by verifying whether their geo-replicated versions obey the serializable semantics of the non-replicated application. Rigi takes application codes as inputs and infers a set of side effects and path conditions possibly leading to consistency violations. Rigi employs the Z3 theorem prover to identify pairs of conflicting side effects and feed them to a geo-replication framework for automated across-site deployment. We evaluate AutoGR by transforming four serializable and originally non-replicated DB-compliant applications to geo-replicated ones across 3 sites. Compared with state-of-the-art human-intervention-free automated approaches (e.g., strong consistency), AutoGR reduces up to 61.8% latency and achieves up to 2.12× higher peak throughput. Compared with state-of-the-art approaches relying on a manual analysis (e.g., PoR), AutoGR can quickly enable the geo-replication feature with zero human intervention while offering similarly low latency and high throughput.

PVLDB Reference Format:

1 INTRODUCTION
To offer fast responses for users worldwide [2, 5, 47], many Internet services replicate data across multiple geographically dispersed data centers [23, 24, 26, 40, 50]. However, geo-replicated services face an expensive wide-area communication problem, since concurrent user requests across sites must be coordinated for precluding incorrect behaviors, e.g., states may permanently diverge between sites, and/or application-specific invariants may be violated.

Various fine-grained consistency models have been proposed for safely avoiding unnecessary coordination [30, 36, 42, 56], PoR consistency [36] and Generic Broadcast [42] express the consistency semantics as the ordering restrictions over pairs of operations so that weakening or strengthening the consistency semantics can be achieved by imposing fewer or more restrictions. Tools like Indigo [20] and SIEVE [34] can assist developers in identifying necessary coordination, though it requires programmers to write application-specific invariants and code specifications [18]. Such a manual process is often very time-consuming and error-prone.

In this paper, we present an end-to-end framework named AutoGR (Automated Geo-Replication), which automatically deploys geo-replicated applications based on their non-replicated serializable versions without manually dealing with restriction-based fine-grained consistency. AutoGR requires zero programmer efforts to write any form of specifications while minimizing cross-site coordination costs and preserving correct application semantics.

Our work is based on the key observation that the side effects produced by user requests and the application-specific invariants that programmers need to maintain are already implicitly reflected in the logic of the programs. Therefore, it is possible to automatically infer through a static program analysis of the code behaviors and the invariants to detect potential conflicting operations that need to be coordinated. For instance, to implement the withdraw operation of a banking system, we must subtract from the current balance a delta, which is specified by users. This subtraction summarizes the intended code behaviors. To make this subtraction happen, we must check the code to make sure the amount to be withdrawn should be

---

The diagram in Figure 1 illustrates the framework of AutoGR. It consists of an Analyzer (Rigi) that takes application codes as inputs and infers application-specific invariants. These invariants are then used to coordinate geo-replication across multiple sites efficiently. The static and runtime components of the framework work together to ensure both performance and correctness.

---
less than or equal to the current balance. This checking reflects the intended invariant that the balance must always be non-negative.

Following this insight, we propose, as one of our key contributions—a novel static analyzer called Rigi. As shown in Figure 1, Rigi forms the static part of the AutoGR framework. In this paper, we focus on the DB-compliant applications that implement their logics in high-level language such as Java and store their states in transactional databases. Rigi takes the non-replicated, serializable application code and the corresponding database schema as inputs. Without requiring any form of user specifications, it traverses the corresponding control flow graph and extracts the side effects of each operation over the shared data and the corresponding path conditions that need to be met before the effects can be made to ensure correct application semantics. With the help of our formal database abstraction, we make those side effects and conditions verifiable by further translating them into the code of Z3, a popular and powerful theorem prover [27].

Rigi then employs Z3 to automatically check the commutativity of the side effects of every pair of operations, and the compatibility of their corresponding path conditions. If pairs of conflicting side effects are detected, Rigi generates ordering restrictions as its outputs. The ordering restrictions are enforced by the runtime of AutoGR to ensure the geo-replicated execution obeys the serializable semantics of the non-replicated version of the application [41].

AutoGR integrates Rigi with an existing geo-replication and coordination system Olisipo [7, 36], see Figure 1. The runtime of AutoGR automatically deploys the geo-replicated application through carefully coordinating the operations confined by the restrictions generated by Rigi between sites. This guarantees that the generation of their conflicting side effects are serialized, and these effects are propagated and replicated across all sites according to the determined proper orders.

To demonstrate the power of AutoGR, we transform four non-replicated applications including SmallBank [53], RUBiS [29], Seats reservation [54], and HealthPlus [6] into their geo-replicated versions. Our static analysis results suggest that the automatic reasoning cost is low and it is able to find the same minimal set of restrictions as state-of-the-art solutions that need extensive efforts from the programmer (e.g., PoR consistency [36]). The experimental results with a 3-site geo-replication setting highlighted that compared with state-of-the-art human-intervention-free approaches (e.g., strong consistency), AutoGR reduces up to 61.8% latency and achieves up to 2.12× higher peak throughput; compared with manual analysis using state-of-the-art approaches (e.g., PoR), AutoGR can quickly enable geo-replication with zero human intervention while offering similarly low latency and high peak throughput.

We summarize our main contributions below:

- To the best of our knowledge, AutoGR is the first end-to-end geo-replication framework that can automatically transform non-replicated, serializable applications into their fast geo-replicated version with little user interventions and without compromising important system properties such as state convergence and invariant preservation.
- We offer an analyzer tool Rigi for performing static analysis over source codes written in Java with manipulating data via SQL interface. Rigi suggests a minimal set of ordering restrictions over pairs of conflicting operations, requiring zero efforts from developers. The core of Rigi consists of the database abstraction in Z3, covering a substantial range of SQL features, and the general commutativity and semantics checks.
- We evaluate AutoGR on three widely used benchmark applications and a real-world large web application to demonstrate AutoGR’s generality and practicability for a wide range of applications, and its strengths over existing works in automatically enabling the geo-replication feature with superior performance on application latency and system throughput.

2 PRELIMINARIES

2.1 System model

We assume a distributed and replicated system consisting of multiple replicas, spanning over geographically dispersed data centers (a.k.a. sites). User requests (a.k.a. operations) are first submitted to a nearby replica (denoted as primary) and are executed on that replica against its local state $S$. As the first step, this execution only performs condition checks (taking banking as an example, a withdraw request checks whether the amount of money requested is greater than 0 and the balance is sufficient) and identifies the corresponding side effect, but does not commit it. We model a side effect as a deterministic function, which takes some arguments as the input and transition the system from one state to another. Upon completion of the primary execution, the collected side effect is then propagated and replicated to all replicas, including the primary one. Note that the state at a site that a side effect is applied to might differ from $S$ observed when the corresponding side effect was created. Following this, all sites together establish a global partial order on the generation of side effects. Each site applies those side effects in a total order, which may differ from site to site but must be compatible with the global partial order.

2.2 Restriction-based consistency model

Data copies at different replicas must be synchronized for maintaining “consistency.” Traditionally, consistency means serializability [21], i.e., the results obtained by concurrently executing a set of operations/transactions should be equivalent to one of some serial orders. However, the synchronization cost across wide-area nodes is extremely high; moreover, a few industrial studies point out even a slight increase in user experienced latency can lead to significant revenue loss [2, 5, 47]. To offer low latency responses, several fine-grained consistency models have been proposed towards minimizing the expensive cross-site coordination in geo-replicating services [17, 39, 45, 46, 52]. More recently, PoR consistency [36], expresses consistency semantics as a set of ordering restrictions (for short, restrictions) over pairs of operations. Formally, for any two operations $u$ and $v$, if there exists a restriction over them, then $u$ and $v$ must be serialized so that their side effects are produced and executed on all sites following the same order. For example, in a banking system, to avoid over-withdrawal, one has to impose a restriction over any pair of withdraws.

The key ideas behind the above proposals are to make the behaviors delivered by the concurrent geo-replicated executions be explained by some serial executions. In particular, they guarantee that the following two crucial system properties are maintained under relaxed consistency models. Informally, the first property
is called state convergence, i.e., when beginning at the same initial state, after applying the same set of side effects (possibly in a different order), all sites reach the same final state. Besides, the second property is called invariant preservation, which says that at each site, every state transition triggered by applying a side effect should not violate application-specific invariants defined by developers, assuming the initial state preserving those invariants (We call invariant-preserving state valid).

For geo-replicating an application, the key to making the best use of the restriction-based fine-grained consistency models is to identify all pairs of conflicting operations, each of which will be confined by a restriction. AUTOGR is built atop of this line of work.

2.3 Tool support and limitations

Some works have formulated a set of principles to guide developers to identify restrictions [17, 20, 32, 36]. However, most of them require extensive domain expertise and manual efforts to reason about all possible concurrent executions of user requests. Such a manual reasoning process is time-consuming and error-prone.

To relieve the burden imposed on programmers, many attempts have been made towards offering automated tools for completing this task [20, 34]. SIEVE computes the weakest preconditions for each side effect, which summarizes when they cannot be executed without coordination [34]. This solution is conservative, as it requires coordinating the generation and replication of all problematic side effects, whose weakest preconditions are evaluated to False. To improve the limitation of SIEVE, Indigo performs static analysis of operation post-conditions against invariants to infer the pairs of concurrent operations that may lead to invariant violations [20]. One of the major drawbacks of this line of work is that they all require programmers to write correct and sufficient specifications about application-specific invariants and pre- or post-conditions of their code in logic formulas. Another major drawback is that they assume that the specifications written by programmers are always correct and sufficient. Incomplete or too weak invariants can cause incorrect system behaviors, while too strong or unnecessary invariants lead to performance penalty. As a consequence, it’s desired to completely release this burden from programmers by not requiring them to specify application-specific invariants.

2.4 The Z3 theorem prover

Z3 is a state-of-the-art theorem prover developed by Microsoft Research [27], and has been widely used to check the satisfiability of logical formulas [37, 49]. Those formulas are input constraints given by users and can be written as a Boolean combination of atomic formulas, which are defined over a rich set of theories supported by Z3, such as integer and real arithmetic, bit-vectors, arrays, quantifiers, and functions. Given a formula in the first-order logic calculus with free variables, Z3 searches a set of assignments to its variables, which satisfy that formula, and reports unsat, if the search fails. We can take advantage of the rich set of theories in Z3 to model the changes to the shared state of target applications, while using the solver to reason about the consistency constraints encoded in the original code.

```java
void withdraw(Connection conn, String custName, double amount) throws Exception {
    PreparedStatement stmt = conn.prepareStatement("SELECT * FROM ACCOUNTS WHERE name = ?");
    stmt.setString(1, custName);
    ResultSet rs = stmt.executeQuery();
    if (rs.next()) {
        double balance = rs.getDouble(1); // custbalance
        long custId = rs.getLong(1); // custId
        stmt = conn.prepareStatement("UPDATE SAVINGS SET bal = ? WHERE custid = ?");
        stmt.setDouble(1, balance); // custbalance
        stmt.setLong(2, custId); // custId
        stmt.executeUpdate();
        conn.commit();
    }
}
```

Figure 2: Pseudocode of a Withdraw transaction

3 OVERVIEW

The goal of AUTOGR is to develop an end-to-end framework to automatically deploy non-replicated serializable code in a geo-replicated setting to explore performance benefits while maintaining the correct application semantics. The performance gains enabled by AUTOGR come from two parts. First, AUTOGR leverages on an existing geo-replication framework OLISIPO that enables fine-grained coordination over pairs of operations that produce conflicting side effects. Second, AUTOGR integrates OLISIPO with a static analyzer Rigi to identify a minimal set of ordering restrictions that must be ensured so that the intended semantics of the corresponding geo-replication applications are not violated.

3.1 Target applications

We target and transition classic three-tier non-replicated applications [1, 10]. One of such applications spans its logic across client, application server, and database tiers, and assumes a single-copy database (often a relational database). The application server contains the functional business logic, defined as a set of transactions. Each transaction takes requests from clients as inputs, and intellectually manipulates the state stored in the database via SQL queries. These queries are statically defined as parameterized functions, and instantiated by user inputs at runtime.

Here, we briefly elaborate on how the applications work. When a transaction starts, it takes inputs from users (e.g., the key to the target objects), and executes a SELECT query to retrieve shared objects from the backend relational database. Then, it performs certain condition checks to determine the side effects. For instance, in Figure 2, the Withdraw transaction checks the existence of the target account and the balance in that account fetched from the database is greater than or equal to the amount supplied by users. Depending on the checking results, the execution of that transaction can go to different branches, which leads to different side effects. For instance, if the check fails, then the Withdraw transaction generates no side effects. Otherwise, it may perform changes to the shared objects and write those side effects back to the database by executing
update queries such as INSERT, UPDATE, and DELETE. In the end, it commits all side effects and makes the changes persistent.

### 3.2 A motivating example

Next, we use a simplified banking system as a motivating example to explain the design principles behind Riti for identifying conflicting operations and generating ordering restrictions. The system has three operations, namely, Withdraw, Deposit, and UpdateCustomer. The pseudocode of Withdraw is shown in Figure 2, and we omit the other two in the interest of space. Figure 3 gives three concurrent executions of these operations with no coordination placed.

Figure 3a illustrates an example of replicating non-commutative side effects. Two concurrent requests of the UpdateCustomer operation are submitted to two data centers to change the location of “Bob” from “NYC” to “LON” and “FRA,” respectively. When the side effects of the two requests were replicated across the two sites in different orders, the resulting states would differ. This violates a critical system property called state convergence [43]. To preclude this kind of anomalies, we need to check the commutativity of side effects produced by operations in a pair-wise fashion and add a restriction to coordinate the generation of any pairs of side effects that do not commute.

Figure 3b shows an example using the side effects of two concurrent Withdraw requests. Although the two side effects commute, this execution is still invalid. This is because the path conditions established at the primary replica could be invalidated by the concurrent execution of other operations. In this example, the path condition for the side effect of a successful withdrawal would be balance >= money >= 0. Clearly, the condition holds in the primary replica of the two simultaneous Withdraw requests. Nevertheless, it becomes false when the side effect reaches the other (remote) site. Application of the side effects of both requests results in a negative balance value, violating the expected invariant. To preclude this type of violations, we need to derive a semantics check to determine if we should add a restriction between the pair of side effects that are not invariant preserving.

Unlike the above two harmful concurrent executions, Figure 3c represents a correct one, where a Withdraw can run concurrently with a Deposit, for which the reason is two-fold. First, their side effects are commutative. Second, the conditions corresponding to the two identified side effects are compatible, i.e., the side effect of Deposits will not invalidate the condition of successful Withdraws, and vice versa. This indicates that no coordination should be placed to restrict the generation and replication of the two side effects.

### 3.3 Challenges and opportunities

Inspired by the examples shown in Figure 3, the correctness requirement implies that the geo-replicated version should not generate more states than the original, non-replicated version. Therefore, we may need to output a restriction over a pair of side effects if they meet either of the following two conditions: a) they are non-commutative side effects and may cause replicas to terminate in a divergent state that cannot be generated in any serializable order; and b) the condition leading to one of the two side effects might no longer hold when seeing the other side effect.

To detect the two cases without the programmer’s inputs, we need to “understand” the operations solely from the source code to infer side effects and invariants. First, due to the code structure of the target applications, for inferring side effects, it is sufficient to understand the specifications of SQL queries and the changes induced on arguments that instantiate the queries. Second, application-specific invariants are implicitly encoded into the conditional checks taken at each branch along a control flow path of each transaction. As a result, to automatically infer the invariants, we need to extract and understand the semantics of the path condition leading to a particular side effect. This step requires traversing all distinct code paths. It may also involve the understanding of the semantics of SELECT queries, since the variables in the branch conditional check formula may be fetched from the database.

Identifying conflicting pairs of side effects requires us to make the gathered side effects and path conditions verifiable. Here, we translate them into the code of Z3 [27], and then ask Z3 to check the commutativity of the side effects and the compatibility of their
As illustrated in Figure 4, Rigi is comprised of a database specification library (or short, DBLib), a side effect collector (or short, Collector), and a rule checker (or short, Checker).

1. DBLib represents the abstraction of relational databases in Z3Py, a Z3 API in Python. It covers a substantial range of SQL features and serves as a building block for Rigi.

2. Collector inspects the source code and identifies the side effects generated in different code paths. It gathers the path conditions leading to particular side effects. It also translates the side effects and their path conditions into code in Z3Py.

3. Checker calls the Z3 theorem prover [27] and generates restrictions through commutativity and semantics checks over pairs of operations' side effects and their path conditions.

### 4.1 Database abstraction in Z3

Table 1 shows DBLib, our database abstraction in Z3, covers a substantial range of SQL features. The basic features cover the table schema definition and simple queries such as SELECT, UPDATE, etc.

The first row of Table 1 shows the translation of a table definition in SQL into an extensional array in Z3Py, which consists of pairs of \(< KEY, VALUE >\), where KEY represents the unique index of the corresponding table that may include one or more attributes, and VALUE covers the remaining attributes. Both KEY and VALUE are defined as an algebraic data type in Z3Py where each of their attributes is declared as a primitive type such as IntSort, StringSort, BoolSort, and RealSort (We also enable to use FPSort instead of RealSort for stricter floating-point verification). For instance, the primary key of the table RSVN is translated into a customized data structure of type K_RSVN to include two attributes, namely R_C_ID and R_F_ID. Considering that the data for each row of the table can be fetched by a primary key or a unique index key, we additionally allow both of them to point to the same row of the table.

The second four lines of Table 1 represent the translation of basic SQL queries. First, the SELECT and UPDATE queries are translated into Select and Store operations on Z3Py arrays, parameterized by the primary key. We additionally allow specifying which set of attributes of the target record a query intends to fetch or update. Regarding INSERT, we create a key value pair, and make the corresponding key point to the value through the Store function. For DELETE, we just set the target record to be nil.

The last three lines of Table 1 show how we model the advanced SQL features, ranging from comparison operator to aggregation. First, for the comparison operator, we fetch the target attribute by the corresponding key and compare it with the supplied argument, e.g., end_date in the example. Second, we translate cross-table join queries into multiple sub-selections, each of which fetches records of interest from a particular table with the same unique key. Third, for ORDER BY on an attribute, we specify a relation over the list of records that for any adjacent pairs of records, the value of the target attribute of the former record is always either greater or less than the counterpart of the latter record.

### 4.2 Collecting side effects and conditions

The mission of Collector is to collect side effects and their corresponding path conditions, and to translate them into Z3Py code, on a per path basis. It first builds an Abstract Syntax Tree (AST) from the source code of the target application to obtain the control flow graph. Then, it traverses the entire control flow graph to identify paths via a depth-first-search algorithm. The path analysis faces an infinite path problem, in the presence of loops. To address this challenge, we take inspiration from an existing work [34] that many real-world applications (including those we analyze) satisfy a loop iteration independence property, which says that the parts of the state modified in each iteration are disjoint. In this case, we handle loops by first verifying this independence property and then unfolding the independent loops so that the number of paths becomes finite. We explain the handling of complex loops in Section 4.3.

We skip paths that are either read-only or leading to abort, since they do not generate side effects. Then, we iterate all remaining code paths, and collect the side effect for a path by performing one-to-one direct translation from Java code to Z3Py code with the following two exceptions. First, we ignore control flow condition checks along the target path, which will be handled by the condition collecting phase. Second, for every encountered SQL statement, we translate it into its corresponding form using our extended DBLib APIs, presented in Table 1. It is worth mentioning that this one-to-one translation is feasible since Z3 itself already supports common data types and the associated computations, while our DBLib covers the database statements. Take `Wi thdraw` for an example. To update the account balance, the transaction assigns a local variable `balance` by the value fetched from the backend database. Then it performs a local computation to subtract `amount` from `balance`, where `amount` is supplied by the user. At the end, the transaction sets the final value of `balance` to instantiate a `UPDATE` query "update set bal = ? where custId = ?". After the translation, the resulting Z3Py code preserves the semantics of the original code. This translation enables our Z3-based Checker to infer the numeric changes on the `balance` variable, i.e., `decrement` here, to extract commutative side effects.

In addition to side effects, Rigi gathers the path condition by making conjunction of the Boolean formulas of every branch taken by the corresponding path. The individual formula may involve both shared objects fetched from the database or arguments from user inputs, which requires Rigi to substitute the variables involved in each formula with their source. For example, in the banking system, a formula "balance - amount >= 0" determining whether a withdraw can be admitted, is translated into "table_accounts(userId),bal...
Table 1: Specifications in Z3Py automatically derived for database APIs which cover a substantial range of SQL features.

<table>
<thead>
<tr>
<th>Category</th>
<th>Description</th>
<th>Example SQL statement</th>
<th>Z3Py specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data model</td>
<td>CREATE TABLE RSVN (R_C_ID BIGINT NOT NULL, R_F_ID BIGINT NOT NULL, R_SEAT BIGINT NOT NULL, ... PRIMARY KEY (R_C_ID, R_F_ID))</td>
<td>K_RSVN = Datatype(...) K_RSVN.declare(...,(R_C_ID, R_F_ID)) V_RSVN = Datatype(...). V_RSVN.declare(...)</td>
<td>TABLE_RSVN = Array(..., K_RSVN, V_RSVN)</td>
</tr>
<tr>
<td>Basic queries</td>
<td>SELECT NAME FROM ACCOUNTS WHERE CUSTID = cid</td>
<td>V_ACCOUNTS.NAME(SELECT(...))</td>
<td></td>
</tr>
<tr>
<td>Advanced features</td>
<td>SELECT INTO USERS (ID,FIRSTNAME, LASTNAME,...) VALUES (uid,fname, lname,...)</td>
<td>Store(TABLE_USERS, K_USERS.new(uid), V_USERS.new(uid,fname, lname,...))</td>
<td></td>
</tr>
<tr>
<td>Inner join</td>
<td>UPDATE CUSTOMER SET C_BASE_AP_ID = aid, LOCATION = loc WHERE C_ID = cid</td>
<td>Store(TABLE_CUSTOMER, T_CUSTOMER.new(cid), V_CUSTOMER.new(V_CUSTOMER.BALANCE(Select(TABLE_CUSTOMER, T_CUSTOMER.new(cid)) = aid,...,loc,...)))</td>
<td>Store(TABLE_ACCOUNTS, new(cid), V_ACCOUNTS.nill)</td>
</tr>
<tr>
<td>Comparison operator</td>
<td>SELECT ID FROM ITEMS WHERE ID=id AND END_DATE&lt;end_date</td>
<td>V_ITEMS.END_DATE(SELECT(...))</td>
<td>V_ITEMS.END_DATE(SELECT( TABLE_ITEMS, K_ITEMS.new(id)) &lt;= end_date)</td>
</tr>
<tr>
<td>Aggregation</td>
<td>SELECT BALANCE,F_SEATS_LEFT, R_ID, R_SEAT, R_PRICE FROM T_CUSTOMER, T_FLIGHT, T_RSVN WHERE F_ID=aid AND C_ID=R_C_ID AND F_ID=aid AND F_ID=R_F_ID</td>
<td>V_CUSTOMER.BALANCE(Select(T_CUSTOMER, K_CUSTOMER.new(cid))) V_FLIGHT.F_SEATS_LEFT(Select(T_FLIGHT, K_FLIGHT.new(Fid))) V_RSVN.R_ID(Select(T_RSVN, K_RSVN.new(cid,fid))) V_RSVN.R_PRICE(Select(T_RSVN, K_RSVN.new(cid,fid)))</td>
<td>V_RSVN.R_PRICE(Select(T_RSVN, K_RSVN.new(cid,fid)))</td>
</tr>
</tbody>
</table>

As depicted in Figure 4, Checker takes the database specifications and the (o, pcond, arg) tuples generated by Collector as input, and performs two checks, namely the commutativity and semantics check, to figure out the conflicting side effect pairs.

Commutativity check. For two operations \( u_1 \) and \( u_2 \), assume their side effects and arguments are \( (o_1, arg_1) \) and \( (o_2, arg_2) \), respectively. The commutativity check is relatively straightforward and well studied in the literature [25, 55], and concentrates only on side effects. Here, we check, for any reachable state \( s \) of a target application, whether the following assertion is satisfied:

\[
o_1(o_2(s, arg_2), arg_1) = o_2(o_1(s, arg_1), arg_2)
\]

This assertion says that two side effects are commutative if applying them against the same initial state in different orders leads to the same final state. Take the problematic concurrent execution shown in Figure 3a, for example. The side effects of \( \text{UpdateCustomer} \) are not commutative; thus, \( \text{Rigi} \) adds a restriction over any pair of \( \text{UpdateCustomer} \) operations.

Additionally, for a transaction containing complex loops with non-independent iterations, it is hard for \( \text{Rigi} \) to infer their side effects. In this case, we resort to a conservative but safe strategy, where the corresponding transaction will be marked as conflicting with all other side effects that have write-write conflicts with it.

Semantics check. To ensure the transition from a non-replicated serializable code into its geo-replicated version is correct, we perform semantics check by asserting for any concurrent execution of two operations \( u_1 \) and \( u_2 \), the application of their side effects against a valid state in a serial order will not lead to an unreachable state in the non-replicated version. Formally, let \( o_1 \) be the side effect of \( u_1 \), \( o_2 \) and \( o'_2 \) be the side effect of \( u_2 \) generated when \( u_2 \) misses or sees \( o_1 \), we should assert that the two operations \( u_1 \) and \( u_2 \) can be executed without restriction if (1) \( o_2 \) is a no-op side effect (i.e., missing the side effect of \( u_1 \), \( u_2 \) produces no side effect); or (2) \( o_2 = o'_2 \) (i.e., \( u_2 \) produces the same side effect regardless of the existence of \( u_1 \)). Finally, we translate the above principle into checking the following formula:

\[
\text{For all } (s), (s, arg_2) \text{ satisfying } pcond_2 \Rightarrow o_1(s, arg_1), arg_2 \text{ satisfies } pcond_2. 
\]

We use two following counter-examples to illustrate this idea. First, as shown in Figure 3b, imagine we run two withdraws simultaneously against an initial state (balance is 100) where they want to deduct 50 (denoted as \( u_1 \)) and 60 (denoted as \( u_2 \)) from the shared account, respectively. If \( u_1 \) sees the side effect of \( u_2 \) (the remaining balance is 40), then the side effect of \( u_1 \) will never be generated. In this case, without ordering the execution of these two operations may lead to violations in the intended application semantics that balance should never be negative. Thus, \( \text{Rigi} \) automatically adds a restriction over \( u_1 \) and \( u_2 \).

Take \( \text{RUBiS} \) as another counter-example. \( \text{placebid} \) is to bid on an item when the corresponding auction is still open, while \( \text{closeAuction} \) terminates an auction and declares the winner. When \( \text{placebid} \) did not see the side effects of \( \text{closeAuction} \), it would continue to bid for an auction that was already closed. As \( \text{placebid} \) may miss
the side effects of \textit{placebid} which issues a higher bid than those observed by \textit{closeAuction}, the winner declaration would violate an intended application semantic where the winner must be the bidder associated with the highest accepted bid. Following a similar logic above, Rigi produces a restriction over \textit{placebid} and \textit{closeAuction}.

Furthermore, a transaction containing complex loops with non-independent iterations is marked conflicting with all other side effects whose generation phases need to read objects written by it.

In summary, Rigi faithfully follows the program logic of the non-replicated serializable code and may create false positives if the side effect analysis becomes conservative and less accurate, due to complex loops. False positives will negatively impact performance, but they will not cause state divergence or invariant violations. In contrast, it produces no false negatives since we assume that the non-replicated code to be analyzed must be serializable.

4.4 Correctness

To ensure our proposal is correct, we formulate the correctness into the following two theorems, and provide only the proof sketches, in the interest of space.

\textbf{Definition 1 (Idleness).} A \(k\)-site geo-replicated system is idle, if \(k\) sites begin at the same valid initial state, and after operating for a while, all side effects ever generated are applied at all \(k\) sites while no new side effects are being generated at the same time.

\textbf{Theorem 2 (State convergence).} For a \(k\)-site geo-replicated system beginning at the same valid initial state, if the generation and application of side effects is coordinated by the restrictions identified by Rigi and executed by AUTOGR, then when the system is idle, the reached final states of all \(k\) sites converge.

\textbf{Proof.} Assume \(n\) side effects are generated in the target geo-replicated system, denoted by \(o_1, o_2, \ldots, o_n\). Let \((o_1, o_2, \ldots, o_n)\) denotes the state obtained by applying these \(n\) side effects sequentially against the initial state. We show by induction on \(n\) that for all \(n\), all traces of \(n\) side effects are either not admitted by the target due to the Rigi’s static checks and AUTOGR’s runtime coordination, or will lead to the same final state when the system is idle.

(1) \(n = 0\). Trivially convergent.

(2) \(n = 2\). There are two possible traces, \((o_1, o_2)\) and \((o_2, o_1)\). If \((o_1, o_2) \neq (o_2, o_1)\), then one of the two traces will be eliminated by Rigi as the two side effects are not commutative.

(3) For case \(n + 1\), we assume one admissible trace is \((o_1, o_2, \ldots, o_{n+1})\). We then consider an arbitrary trace \((p_1, \ldots, p_{n+1}, p_r, \ldots, p_l)\), where \(r = l + 1\) and the sequence \((p_i)\) is an arbitrary permutation of the sequence \((o_j)\) for \(j \leq n\). If \(l = n\), by the induction hypothesis, the proposition holds. If \(l < n\), for \((p_1, \ldots, p_{n+1}, \ldots, p_n) = (o_1, \ldots, o_{n+1})\) to hold, the only possibility is to move \(o_{n+1}\) to the end. And this condition is only satisfied when \(o_{n+1}\) commutes with all \(p_i\) for \(i \geq r\), which can be shown by induction on \(n\).

\textbf{Theorem 3 (Invariant preservation).} For a \(k\)-site geo-replicated system with a valid initial state, if the generation and application of side effects are coordinated by the restrictions identified by Rigi and executed by AUTOGR, then at each site, every state before generating or after applying a side effect must be valid.

\textbf{Proof.} Let \(I\) denote the application-specific invariants encoded in the non-replicated, serializable application. For a side effect \(o\), we say a state \(s\) is \(o\)-compatible, if \(o\) can be generated against state \(s\), i.e., \(s\) satisfies both \(I\) and the path condition leading to \(o\). For a \(k\)-site geo-replicated version of the target non-replicated application, we need to reason if the states before (denoted by \(s_2\)) and after (denoted by \(s'\)) applying an admitted side effect \(o_1\) are both valid. Without loss of generality, we perform the following analysis against a site \(A\), which receives two side effects \(o_1\) and \(o_2\) from any site in the geo-replicated system (\(A\) itself included). There are three major cases to be considered.

(1) \(o_2\) arrived at \(A\) after \(o_1\), and \(s_2 = s'_1\) (\(o_2\) is right after \(o_1\)).

(a) The generation of \(o_2\) sees the side effect of \(o_1\), and thus \(I\) holds at \(s_1, s_2, s'_1\) by the correctness of the non-replicated version.

(b) The generation of \(o_2\) misses the side effect of \(o_1\). In this case, \(s'_1\) must be \(o_2\)-compatible, by the logic of Rigi’s semantics checks.

(2) \(o_2\) arrived at \(A\) after \(o_1\), but there are several side effects \(o_3, o_4, \ldots, o_1, \ldots, o_{n+2}\) between \(o_1\) and before \(o_2\).

(a) The base case \(n = 0\) is exactly the first case.

(b) For case \(n + 1\), by the induction hypothesis, \(s'_1\) must be \(o_2\)-compatible, and for each \(2 < j < n + 2\), \(s_j\) is \(o_j\)-compatible.

(3) \(o_1\) arrived at \(A\) after \(o_2\).

We prove this by a similar logic in the above case (1) and (2).

By the case analysis above, we therefore show that \(I\) is preserved at every moment before and after applying a side effect. □

4.5 Optimizations

Here, we extend Rigi to incorporate the following advanced features for practical considerations.

\textbf{CRDTs support.} Conflict-free Replicated Data Types (CRDTs) [44, 48] design operations that commute by construction. Thus, applications can leverage CRDTs to increase the space of commutative operations/side effects, in order to relax strong consistency constraints imposed by potential state divergence in the presence of uncoordinated concurrent request executions. To cope with the increasing attention on CRDTs [14, 16], we adapt Rigi to work with CRDTs. This adaption only requires us to extend Collector to generate commutative side effects according to the target CRDT’s intended rules for merging concurrent updates, while leaving the commutativity checks to remain the same.

\textbf{Uniqueness.} The non-replicated, serializable applications often make use of the database’s AUTOINCREMENTAL feature to ensure that a unique number is generated automatically whenever a new record is inserted into the table. When migrating to geo-replication, restrictions must be placed between pairs of side effects, each of which needs to put a record into the same database table, to preclude possible commutativity and semantic violations. To avoid the costly cross-site coordination for the unique id assignment, popular web
services choose to partition the key space and assign different ranges to different replicas [4]. To support this feature, we add another optional axiom to Checker saying that the id obtained in the corresponding side effects is unique, i.e., ‘Vid’, id ≠ id’.

Developers can enable the commutative transformation by turning on the −DOpt option. Note that although these transformations introduce new behaviors, i.e., some serializable execution orders previously rejected by the corresponding non-replicated application are now accepted, theorems 2 and 3 are still valid.

4.6 AutoGR integration
The implementation of Rigi consists of 2.1 k lines of Java code, 3.4 k lines of Kotlin code, and 521 lines of Z3Py code, where the Z3Py code performs the commutativity and semantics checks shared across applications. Rigi uses a Java parser [9] and a SQL statement parser [8] to generate database specifications, analyze the code paths, and collect side effects plus path conditions, for applications that are written in Java and using the SQL interface for data access. Note that we specialize our implementation on Java web applications since Java is one of the most popular languages used for implementing back-end server applications in major Internet service providers [13]. We further take advantage of the fact that the application code is often straightforward with few advanced programming features, thus considering only a subset of Java, e.g., control flow constructs (if, while, try-catch, etc.), built-in data types (Int, String, etc.), method calls, and basic OOP features.

We integrate Rigi with Olisipo, an existing geo-replication and coordination system built by Li et al. [7, 36], to form our end-to-end Geo-Replication AutoGR framework, see Figure 1. Olisipo consists of three key system components. Firstly, there is a runtime library attached with application servers deployed at each site for communicating with the rest of Olisipo for extracting runtime side effects through local transaction execution, and for initiating cross-site replication and necessary coordination. Second, the global coordination service of Olisipo consists of a set of servers, which are spanning over different sites, running a BftSmart [51] consensus protocol. This service will tell whether a side effect that needs coordination can be admitted or not. If so, the dependencies are determined, which implies that this operation is admitted to the global partial order. Otherwise, the coordination service suggests aborting this operation due to conflicts. Finally, at the low level of the runtime there is a geo-replicated data store with replicas spanning over different sites. The store offers causally consistent replication, in which it receives the side effects with ordering dependencies and replicates the side effects at every replica when the specified dependencies are locally satisfied. This replication strategy ensures that the serial order of applying side effects at every site is compatible with the global partial order of the corresponding operations, as pointed by the system model description presented in Section 2. However, there is no guarantee from the geo-replicated store that the final state will converge and the correct application semantics are always preserved. This is all done by the joint work of the coordination layer and the static analyzer Rigi.

This integration requires Rigi to generate a static and unique signature, for each side effect that requires coordination, and the list of static signatures will be consumed by the runtime library for initiating the coordination procedure. In addition, the restrictions identified by Rigi are kept and looked up by the coordination service to serialize pairs of conflicting side effects. To minimize the code changes to make target non-replicated applications geo-replicated, the logic of the runtime library is implemented in the database connector. Following this, the code adaption is just to replace the original connector with the one offered by AutoGR. Olisipo offers “Last-Writer-Win (LWW)” and “UniqueIdGeneration” strategies for transforming the non-commutative side effects into commutative ones to enable geo-replicated services to maximize the performance, which match Rigi’s optimizations. When the −DOpt is enabled, AutoGR automatically assigns the logical timestamps to corresponding attributes to take advantage of LWW, and enables UniqIdGeneration on all primary keys associated with the AUTOINCREMENTAL keyword.

Finally, the workflow in the AutoGR runtime is as follows: user requests are sent to a web server at a site in close proximity to them. At the primary site, the server executes the user request against the local database replica to collect side effects. Then, the web server consults the coordination service for admitting or rejecting the request via the runtime library. When accepting this request, the web server sends the side effect associated with the dependencies to the geo-replicated store for applying side effects across sites. Finally, results and decisions are delivered back to the end users.

4.7 Discussions
Application languages. Though the current implementation of Rigi only supports applications written in basic Java, the design principles of Rigi are general and can be extended to support advanced Java features and other languages. Such extensions can be made by enhancing Collector to gradually take more Java constructs and library functions into consideration or use language-specific parsers. To link other language applications plus the analysis results achieved by Rigi to the runtime of AutoGR, we can modify the language-specific database connectors to intercept database calls for creating runtime path signatures, and extracting read/write sets for cross-site replication and coordination, as described above.

Backend storage systems. AutoGR relies on serializability and transactional support offered by the backend storage systems, since the abstraction of transactions eases the consistency reasoning. In addition, the Z3 abstraction over SQL queries already expresses the behaviors of key-value stores since the former covers the latter. However, compared to SQL-compliant databases, the use of key-value stores would lead us to make Collector stronger since the application logic needs to perform computations that can be handled by SQL queries.

Incremental analysis. Currently, Rigi does not assume incremental analysis when facing changes in the target code base. It suggests that upon code changes, the developers would need to run Rigi against the whole code base again to infer possible new restrictions for safety. This would result in additional efforts to link the updated application with AutoGR, as newly introduced side effects need to generate unique signatures for runtime lookup. Despite that we believe this overhead is manageable, adapting Rigi to support incremental analysis would be a wise option.

Failure handling. At runtime, we rely on Olisipo to handle failures. The state of the site coordinators is replicated through a
Table 2: Conditions and side effects corresponding to a few simplified examples in the four benchmark applications. Note that the values of parameters such as itemId are part of the corresponding side effect, which are obtained when the side effect is generated at the primary site and encoded into that side effect, thus precluding non-determinism.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Condition</th>
<th>Side effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>RUBiS</td>
<td>PlaceBid: table_item(itemId) := nil ∧ table_item(itemId).closed =⇒ False∧ table_user(uId) := nil ∧ table_item(itemId).max_bid &lt; price</td>
<td>table_bids(bid) = (itemId, uId, price) ∧ table_item(itemId).nb_of_bids ++ ∧ table_item(itemId).max_bid = price</td>
</tr>
<tr>
<td>SmallBank</td>
<td>Send-Payment: amount ≥ 0 ∧ table_checking(sendAcc).balance ≥ amount</td>
<td>table_checking(sendAcc).balance = amount ∧ table_checking(destAcc).balance += amount</td>
</tr>
<tr>
<td>Seats</td>
<td>NewReservation: table_flight(fid) := nil ∧ table_flight(fid).seat_left &gt; 0∧ table_rest(fid, seat) := nil ∧ table_rest(fid, cid) =⇒ nil</td>
<td>InsertReservation(table_res, cid, fid, seat, price) ∧ table_flight(fid).seat_left =⇒ 1 ∧ UpdateCustomer(table_customer, cid, info, price) ∧ AddFrequentFlyer(table_flyer, cid, uId)</td>
</tr>
<tr>
<td>Health-Plus</td>
<td>makeLabAppointment: table_lab_appointment(laid) =⇒ nil ∧ table_lab_test(testId) != nil ∧ table_tmp_bill(patientId) =⇒ nil</td>
<td>InsertLabAppointment(laid, testId, patientId, ...) ∧ InsertTmpBill(hillId, patientId, fee)</td>
</tr>
</tbody>
</table>

Table 3: The basic statistics of applications and the lines of code generated by Rigi for applications.

<table>
<thead>
<tr>
<th>Application</th>
<th>Codebase</th>
<th>#op/</th>
<th>#update_op/</th>
<th>#loop_op</th>
<th>Code generated by Rigi</th>
<th>DB</th>
<th>Cond.</th>
<th>Side effects</th>
</tr>
</thead>
<tbody>
<tr>
<td>SmallBank</td>
<td>2.5k</td>
<td>6/5/0</td>
<td>29</td>
<td>38</td>
<td>119</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RUBiS</td>
<td>9.8k</td>
<td>28/6/0</td>
<td>113</td>
<td>62</td>
<td>191</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Seats</td>
<td>5.0k</td>
<td>7/4/2</td>
<td>267</td>
<td>65</td>
<td>207</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>HealthPlus</td>
<td>15.7k</td>
<td>197/50/40</td>
<td>524</td>
<td>1,113</td>
<td>1,387</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 4: Statistics of checks, where #failed_commu and #failed_sem correspond to the number of failed commutativity and semantics checks, respectively.

<table>
<thead>
<tr>
<th>Application</th>
<th>#checks</th>
<th>#failed_commu</th>
<th>#failed_sem</th>
<th>#restr.</th>
</tr>
</thead>
<tbody>
<tr>
<td>SmallBank</td>
<td>normal</td>
<td>20</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>RUBiS</td>
<td>opt.</td>
<td>90</td>
<td>19</td>
<td>15</td>
</tr>
<tr>
<td>Seats</td>
<td>opt.</td>
<td>90</td>
<td>32</td>
<td>26</td>
</tr>
<tr>
<td>HealthPlus</td>
<td>opt.</td>
<td>5112</td>
<td>148</td>
<td>76</td>
</tr>
</tbody>
</table>

Paxos-like protocol, thus tolerating up to $f$ failures. However, the vulnerability in AutoGR lies in the operation propagation. Currently, AutoGR relies on each site to send its own local side effects to all remote sites. A pair-wise network outage or failure of a site would possibly result in partial replication, e.g., side effects were missing by some sites. However, this can be addressed using standard techniques for exchanging causal logs or reliable multicast.

5 EVALUATION

We exercise four web applications: SmallBank [53], RUBiS [29], Seats Reservation [54], and HealthPlus [6]. The first three have been extensively used in related work and the last one is a real-world deployable application. They are designed by having web servers running application logic to interact with a shared backend database. Table 3 shows their basic statistics. The simplest application, SmallBank, simulates an online banking system where four out of the total five transactions contain updates. RUBiS emulates an eBay-like [3] online auction website, where six out of the total 16 transactions are update transactions. Seats Reservation (or Short, Seats) models an electronic Airline ticketing service. It consists of six transactions, four out of which contain updates. HealthPlus is a real-world deployable management system for health care facility. It consists of 157 transactions in total and 50 of which are updates. Given the large code base and the prohibitive number of checks (5112) for manual analysis in HealthPlus, we use it to evaluate the generality and practicability of our approach.

5.1 Static analysis

Here, we report the Rigi’s main results and refer readers to our online documents [12] for other detailed results.

Side effects and path conditions. Table 3 summarizes the number of code automatically generated by Rigi. First, it reads the database organization files (e.g., SQL schema) to produce the Z3Py specifications about the table data structures. For this purpose, Rigi generates 29, 113, 267, and 524 lines of database specifications for SmallBank, RUBiS, Seats, and HealthPlus, respectively. SmallBank has the least amount of database specifications, as it has only three tables, while HealthPlus is at the other end with 37 tables.

Second, Rigi traverses all code paths, and creates 119-1387 and 38-1113 lines of side effect and path condition specifications for the four applications, respectively. We show the exemplified extracted side effects and path conditions in Table 2. The number of side effect and path condition specifications is proportional to the codebase size for applications except for Seats. Though Seats has fewer tables and transactions than RUBiS, they are more complex (e.g., containing more attributes, or more state changes), leading to 136.28% and 8.83% more DB and side effect specifications, respectively. Finally, we also examine the number of paths and loops in those update transactions. In SmallBank and RUBiS, each update transaction does not contain loops and has only a single path leading to side effects. By contrast, there are 2 and 40 transactions that contain loops in Seats and HealthPlus, respectively. Rigi further discovers that the two Seats’ loops obey the loop iteration independence property, thus can be unfolded one time. For HealthPlus, 24 out of 40 loop transactions obey the independence property. Overall, each of these transactions of the four applications leads to between 1 and 9 pairs of conditions and side effects.

Checks and restrictions. Table 4 summarizes the number of restrictions identified by Rigi for four applications. By taking pairs of
side effects and path conditions extracted above as the input, Rigi performs 20, 90, 90, and 5112 checks for SmallBank, RUBiS, Seats, and HealthPlus, respectively, where half are commutativity checks, and the other half are semantics checks. Based on the pair-wise checking presented in Section 4.3, the number of checks is roughly the square of the number of distinct code paths. This explains the substantial gap in the number of checks between HealthPlus and the other three applications.

We configure Rigi with two modes: Normal or Opt, corresponding to disable or enable the optimizations presented in Section 4.5. Note that switching on optimizations may affect the number of identified restrictions, rather than the number of checks. We first focus on the the Normal mode results. For SmallBank, all side effects of its four update transactions are commutative w.r.t. each other, thus it does not report any commutativity violation restrictions. Besides, it identifies four restrictions related to the failures of semantics checks. Take the restriction between a pair of SendPayment operations as an example. The semantics check performed by Rigi takes as input the path condition and side effect of SendPayment as shown in Table 2, and identifies a possible violation: when two SendPayment operations are running concurrently, the side effect of one operation possibly invalidates a path condition associated with the side effect of the other, which asserts the remaining balance is enough. The remaining three restrictions follow a similar logic.

Rigi suggests 21 restrictions for RUBiS, out of which 19 and 15 correspond to commutativity and semantic violations, respectively. There are 35 restrictions identified for Seats, where 32 and 26 are related to commutativity and semantic violations, respectively. HealthPlus has 148 restrictions, out of which 76 correspond to both commutativity and semantic violations, while the remaining 72 are only related to failed commutativity checks. Interestingly, there are 13, 23, and 76 pairs of side effects and path conditions fail in both commutativity and semantics checks, for RUBiS, Seats, and HealthPlus, respectively. Take the pair (RegisterItem, StoreBid) in RUBiS as an example. Its commutativity check fails because RegisterItem overwrites nb_of_bids, which is increased by StoreBid. Its semantics check fails as StoreBid asserts the new bid is larger than max_bid, which can be invalidated by RegisterItem when inserting a new item. Among the four applications, Seats exhibits the highest ratio of restricted pairs of side effects. The reason is two-fold. First, side effects of Seats’ transactions are complex and with limited commutativity, due to either conflicting writes on shared attributes or concurrent addition and removal from a shared database table. Second, all reservation-related side effects have to ensure a strict invariant, encoded in the corresponding path conditions, that one seat cannot be sold to two different users.

Next, we evaluate Rigi under the Opt mode, which enables the two commutative transforming features introduced in Section 4.5. When switching from Normal to Opt, the results of SmallBank remain unchanged, since all its side effects commute w.r.t each other. The numbers of restrictions of RUBiS and HealthPlus are greatly reduced from 21 to 8, and 148 to 74, respectively. For RUBiS, the UniqueldGeneration feature eliminates 15 and 7 failed commutativity and semantics checks. Similarly, the last-writer-win and UniqueldGeneration strategy together introduce a 58.78% drop in the number of failed commutativity checks for HealthPlus, and UniqueldGeneration additionally eliminates 29 failed semantics checks.

![Figure 5: Analysis Cost breakdown for static analysis of four representative applications. In each bar cluster, the bars on the left and right correspond to the time cost of running Rigi under Normal or Opt mode, respectively.](image)

Nevertheless, we observe the limited improvement for Seats even with optimizations, i.e., only 7 failed commutativity checks have been removed while the failed semantics checks remain the same. This is because only commutativity violations corresponding to the UpdateCustomer side effects can be eliminated by the last-writer-win strategy. We also see the opportunity to apply some other advanced CRDT solutions, e.g., add-win or remove-win set [46], to eliminate the commutativity violations between NewReservation and DeleteReservation. Considering the correctness reasoning of such transformation, we leave this exploration as future work.

**Comparison to other analysis methods.** Here, we compare our work with the most relevant tool Indigo. However, as Indigo offers only two simple APIs (Decrement and Increment), it is impossible to write specifications for the three case study applications. Therefore, to evaluate the effectiveness of Rigi, we choose to compare its results with manual efforts to achieve PoR consistency, which identifies a set of rules to find a minimal set of restrictions [36]. We find that the restrictions automatically identified by Rigi match those found by the manual process for applications except HealthPlus. This shows that Rigi can find a minimal set of sufficient restrictions without requiring any explicit invariants and conditions describing the target operations’ behaviors. However, with more than 5000 checks required by HealthPlus (see Table 4), it is impractical to perform the reasoning manually.

**Performance implications of Rigi.** We measure Rigi’s time cost on a server, which has 16 Intel(R) Xeon(R) CPU E5-2620 cores and 64GB RAM, and runs CentOS 7, Python 3.6.6, Z3 4.8.10, and Java 8. The single-thread performance numbers are summarized in Figure 5 under the two aforementioned modes. Overall, within the Normal mode, Rigi takes from 23.8 seconds to 1.5 hours to analyze all four applications. As expected, it spends the least amount of the analysis time for the simplest SmallBank, and the most for HealthPlus with the largest codebase.

Figure 5 presents the analysis cost breakdown for the four steps of the static analysis: generating DB specifications, collecting side effects plus conditions, checking semantic violations, and checking commutativity. Among the four steps, the time cost of the first two steps is almost negligible. For instance, it only takes 29 seconds to complete the DB specification generation and side effect/path condition even for the largest HealthPlus. The commutativity and
semantics checks dominate the analysis cost and the cost of performing these checks exhibits quadratic growth. Among the four applications, it takes the least time cost to analyze SmallBank, as the side effects and conditions of this application are relatively straightforward, plus it requires the least number of checks. Although Seats contains fewer transactions than RUBiS, it takes a longer time to complete the static analysis. This is due to the following two reasons. First, some transactions in Seats have more paths than RUBiS. Second, some paths of Seats contain more update queries than RUBiS. Thus, the analysis of Seats is more complex than RUBiS. HealthPlus observes the longest analysis time due to its larger transaction space.

When switching to Opt, the time for computing the DB specifications and collecting pairs of conditions and side effects remains almost the same across the four applications. SmallBank observes no change in the checking time since its checks are not affected (see Table 4). However, we observe an up to 58.38% reduction in the commutativity checking time for the remaining three applications. This result is consistent with the elimination of failed commutativity checks in Table 4. Furthermore, the semantics checking time has also been reduced for RUBiS and HealthPlus, except Seats. This is because the semantics checks of Seats remain the same.

Additionally, with 16 threads, Rigi under the Normal mode introduces a 3.2-11.5X speedup of the static analysis time for the four examined applications, e.g., in the largest improvement, it reduces the time cost of analyzing HealthPlus from 5351.209s to 465.402s. Thus, we conclude that given the analysis is a one-time and offline job, the cost is moderate, and Rigi scales well to applications with larger sizes. Note that even for experienced experts with state-of-the-art tools like PoR, it may take a few days for small-scale applications and usually it is infeasible for large ones. Thus the analysis cost saving is tremendous, not to mention that the whole process is fully automated with little manual effort.

5.2 Geo-replication

System configurations. We run geo-replicated experiments on EC2 m4.2xlarge virtual machines (VM) across three sites: Asia-Pacific-Southeast (AP), US-West (US), and EU-Central (EU). Each VM has 8 vCPUs, 32 GB of RAM, and runs Ubuntu 16.04, MySQL 5.5, Tomcat 6.0, and Java 8. The average round-trip latency between any pair of the three sites ranges from 157 to 169ms.

We deploy AutoGR across the three sites, where each site has a database server host a copy of replicated data, a coordinator to participate in a Paxos-like consensus protocol [22], and a proxy that is attached to the original application code to execute user requests via AutoGR. We run applications with two different configurations: (a) we configure AutoGR with the set of restrictions identified by Rigi without the commutative transformation (denoted as “AutoGR-Normal”); and (b) we configure AutoGR with the set of restrictions identified when the commutative transformation is enabled (denoted as “AutoGR-Opt”).

Baselines. We deploy two baselines: (1) a geo-replicated service spanning three sites but serializing all updates, which is a standard deployment offering strong consistency and requiring no manual work, denoted as “Strong”; and (2) a three-site geo-replicated service coordinating use requests guided by the aforementioned manually identified restrictions (denoted as “Manual”).

Datasets and workloads. The dataset of RUBiS is generated using the following parameters: 500,000 old items, 33,000 active items, and 1,000,000 users. We populate a 1.3-2.5 GB database with randomly generated records for the remaining three applications. RUBiS runs the representative bidding mix workload, which consists of 85% read-only transactions and 15% update transactions. For the other three applications, we generate workloads for them by following the RUBiS’s setup. Users are evenly distributed across the three sites, issuing requests in a close loop to the replica at the closest proximity. We measure average user-observed latency per site and the aggregated throughput across sites by adding up their individual values. For each evaluation point, we repeat experiments three times and report the average.

User-observed latency. The primary goal of our proposal is to automatically transition non-replicated applications to be geo-replicated with enhanced performance in terms of low user perceived latency. Figure 6 summarizes the average latency observed by users at different sites across all system configurations. The error bars represent standard deviation. For all applications except HealthPlus, compared to “Manual”, “AutoGR-Opt” achieves similar latency performance. This finding is consistent with the results obtained in the case studies section that Rigi with the commutative transformation option enabled can identify the same restriction set as PoR consistency does. We do not show such a comparison for HealthPlus since the manual analysis is infeasible for such a large codebase. In comparison with “Strong”, “AutoGR-Opt” reduces the user observed latency from 39.8% to 61.8% for all four applications. This is because the set of identified restrictions is minimal and only a small fraction of requests that need cross-site coordination, while in “Strong” all update requests are coordinated. Among the four applications, the latency improvement of RUBiS is the best, while that of Seats is the worst. This is because with the AutoGR-Opt setting, RUBiS imposes the fewest number of restrictions, but Seats needs the most.

“AutoGR-Normal” achieves 24.84% to 54.1% lower user observed latency than “Strong” for RUBiS, SmallBank, and HealthPlus, but behaves similarly as “Strong” for Seats. This result is consistent.
with the restrictions identified by Rigi without the commutative transformation. In that case, "AutoGR-Normal" almost serializes all update transactions for Seats. Finally, "AutoGR-Opt" achieves the same performance as "AutoGR-Normal" for SmallBank and HealthPlus for different reasons. For SmallBank, "Opt" does not eliminate its restrictions. Unlike this, even the number of restrictions drops in HealthPlus, the limited performance improvement is because the workload is random and the number of user requests, whose side effects are restricted by AutoGR-Normal but freed by AutoGR-Opt, accounts for a quite small fraction. Therefore, we expect visible improvements in the presence of more skewed workloads. In contrast, "AutoGR-Opt" reduces latency from 24.8% to 26.0% than "AutoGR-Normal" for both RUBiS.

**Peak throughput.** Next, we shift our attention to the implication of peak throughput. Figure 7 illustrates the peak throughput of different system configurations, which is normalized to the peak throughput of "Strong". These results are consistent with latency observed in Figure 6. "AutoGR-Opt" performs the best for all applications. It achieves as good performance as "Manual", which represents the best performance gain with the most human intervention for manually identifying restrictions, for all applications except HealthPlus (Again, we haven’t conducted the manual analysis for HealthPlus since it can be extremely time-consuming and also error-prone). "AutoGR-Opt" achieves a speedup ranging from 1.39 to 2.12 times, compared to "Strong". Finally, if we disable the commutative transformation, "AutoGR-Normal" still introduces a 1.22-2.12× speedup of peak throughput for all applications except Seats. The reasons for throughput comparison are similar to those of the latency comparison we presented above.

### 6 RELATED WORK

**Geo-Replicated Systems.** Many cloud storage systems offer geo-replication features [11, 15, 24, 26, 28], assuming either strong or eventual consistency [24, 26, 28]. Recently, Azure CosmosDB [15] and Google Cloud DataStore [11] provide a set of APIs that read data with different consistency guarantees, but they still serialize updates for strong consistency. Unlike them, AutoGR follows restriction-based consistency models, and automatically enables the geo-replication feature for un-replicated applications with enhanced performance without requiring developers to identify manually problematic concurrent executions that should be coordinated.

**Consistency models and fine-grained coordination.** To close the gap between strong and eventual consistency, hybrid consistency models, such as RedBlue consistency [35], have been proposed to allow strongly and causally consistent operations to co-exist in a single system. Recently, to minimize the coordination in geo-replication, fine-grained consistency proposals such as PoR consistency [36] completely drop the definition of consistency levels and instead map the consistency semantics into a set of ordering restrictions over pairs of operations (analogous to "conflict relation" in Generic Broadcast [42]). Our approach is built on top of this line of work and extends them for two main aspects – a) we build an automated analysis tool, which identifies the required restrictions and free programmers from the error-prone and time-consuming task; and b) we offer an end-to-end solution for automatically deploying non-replicated serializable applications as geo-replicated with enhanced performance and preserved application semantics.

**Tool supports.** Some recent proposals aim to relieve developers’ burdens to adapt their applications to use fine-grained consistency models in geo-replicated scenarios [17, 30, 34]. SIEVE statically computes the weakest preconditions which lead to side effects that always preserve invariants [34]. Indigo [20] checks whether the concurrent execution of a pair of operations would violate invariants and provides a set of mechanisms to resolve conflicts. Hamsaz takes a sequential object as input and automatically synthesizes a replicated object that avoids unnecessary coordination [31]. These works rely on specifications and application-specific invariants that programmers need to write manually. Unlike them, Rigi does not assume the existence of specifications; instead, it infers the side effects and their path conditions solely from the source code. MixT is a language for designing geo-distributed transactions [36], where different levels of consistency are associated with the attributes of database tables. However, this approach requires programmers to re-implement their applications using MixT. The goal of IPA [19] is orthogonal to ours, as it ensures invariants on an eventually consistent replicated store by repairing and compensating violations introduced by running conflicting operations in parallel.

**CRDTs.** We significantly differ from CRDTs [44, 48] by (1) the primary interest of AutoGR and Rigi is to identify pairs of non-commutative side effects based on the original code rather than improving operation commutativity; and (2) convergence alone cannot prevent invariant violations, which can be avoided by Rigi’s semantics checks and AutoGR’s runtime coordination. Furthermore, CRDTs are complementary to our proposal, and AutoGR can be extended to incorporate more CRDTs, as some case-study applications show limited commutativity and the use of LWW indeed leads to fewer restrictions and better geo-replication performance.

### 7 CONCLUSION

We present AutoGR – an automated end-to-end framework for deploying non-replicated applications as geo-replicated. The core of AutoGR is a static analysis tool Rigi that automatically identifies necessary restrictions without user interventions. Experimental results show that AutoGR has the following benefits: (1) low reasoning cost, (2) optimal performance, and (3) little manual effort.

### ACKNOWLEDGMENTS

We sincerely thank all anonymous reviewers for their insightful feedback. This work was supported by National Key R&D Program of China under Grant No. 2018YFB1003204, National Nature Science Foundation of China (61802358, 61832011, 61922039), USTC Research Funds of the Double First-Class Initiative (YD2150002006), and National Science Foundation (CCF-1756013, IIS-1838024, and CAREER-2048044). Cheng Li is the corresponding author.


