$\mathtt{R}^3$: Record-Replay-Retroaction for Database-Backed Applications

Qian Li
Peter Kraft
Stanford University
{qianli,kraftp}@cs.stanford.edu

Michael Cafarella
Çağatay Demiralp
MIT CSAIL
{michjc,cagatay}@csail.mit.edu

Goetz Graefe
Google
goetzg@google.com

Christos Kozyrakis
Stanford University
christos@cs.stanford.edu

Michael Stonebraker
MIT CSAIL
stonebraker@csail.mit.edu

Lalith Suresh
Feldera
lalith@feldera.com

Xiangyao Yu
UW-Madison
yxy@cs.wisc.edu

Matei Zaharia
UC Berkeley
matei@berkeley.edu

ABSTRACT

Developers would benefit greatly from time travel: being able to *faithfully replay* past executions and *retroactively execute* modified code on past events. Currently, replay and retroaction are impractical because they require expensively capturing fine-grained timing information to reproduce concurrent accesses to shared state. In this paper, we propose practical time travel for *database-backed applications*, an important class of distributed applications that access shared state through transactions.

We present $\mathtt{R}^3$, a novel Record-Replay-Retroaction tool. $\mathtt{R}^3$ implements a lightweight interceptor to record concurrency information for applications at transaction-level granularity, enabling replay and retroaction with minimal overhead. We address key challenges in both replay and retroaction. First, we design a novel algorithm for faithfully reproducing application requests running with snapshot isolation, allowing $\mathtt{R}^3$ to support most production DBMSs. Second, we develop a retroactive execution mechanism that provides high fidelity with the original trace while supporting nearly arbitrary code modifications. We demonstrate how $\mathtt{R}^3$ simplifies debugging for real, hard-to-reproduce concurrency bugs from popular open-source web applications. We evaluate $\mathtt{R}^3$ using TPC-C and microservice workloads and show that $\mathtt{R}^3$ always-on recording has a small performance overhead (<25% for point queries but <0.1% for complex transactions like in TPC-C) during normal application execution and that $\mathtt{R}^3$ can retroactively execute bugfixed code over recorded traces within $0.11 - 0.78\times$ of the original execution time.

PVLDB Reference Format:
Qian Li, Peter Kraft, Michael Cafarella, Çağatay Demiralp, Goetz Graefe, Christos Kozyrakis, Michael Stonebraker, Lalith Suresh, Xiangyao Yu, and Matei Zaharia. $\mathtt{R}^3$: Record-Replay-Retroaction for Database-Backed Applications. PVLDB, 16(11): XXX-XXX, 2023.
doi:XX.XX/XXX.XX

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/DBOS-project/apiary/tree/r3-exp.

Proceedings of the VLDB Endowment, Vol. 16, No. 11 ISSN 2150-8097.
doi:XX.XX/XXX.XX

1 INTRODUCTION

Building and maintaining production applications would be much easier for developers if they could travel back in time. In this context, time travel has two major benefits. The first is *replay*, faithfully re-executing recorded past events in a controlled environment, for example to investigate a rare issue that only occurred in production. The second is *retroaction*, executing new or modified code on past events, for example to test the correctness of a bug fix.

Unfortunately, while there is a rich body of work on replay and retroaction, existing systems are impractical to deploy in production applications. Some record-and-replay systems, such as RR [27], only support single-threaded execution. Others, like Arnold [8], support multi-threaded applications but require expensive and heavyweight instrumentation to track the timings of concurrent accesses to shared state. Moreover, these systems do not support retroaction. Existing retroaction systems have restrictive semantics and do not support concurrent historical executions. For example, Retro-λ [19] supports retroaction for a single-threaded microservice built with the command query responsibility segregation (CQRS) pattern.

In this paper, we present $\mathtt{R}^3$, a novel Record-Replay-Retroaction tool. Our key insight is that time travel is uniquely practical for *database-backed* applications that access shared state through transactions. This is a large and important class of applications, including most microservices, web backends, and serverless applications. Unlike existing systems that record fine-grained timing information for memory or disk accesses to capture data races, $\mathtt{R}^3$ leverages the transaction-oriented state access pattern. It records concurrent state accesses at a coarse transaction-level granularity, enabling faithful replay and retroaction with minimal overhead. We have previously discussed how transactions make debugging easy in a vision paper [15]. In this paper, we design and implement a practical tool for supporting time travel with strong guarantees for database-backed distributed applications. To achieve this, we address significant technical challenges in both replay and retroaction.

The main challenge in replay is supporting widely used production database systems and settings, specifically the commonly used transaction isolation levels. If applications used serializable isolation, we could simply replay transactions sequentially in serial order. However, many applications and popular databases use snapshot isolation to enhance performance. This complicates replay because there may not exist a serial order [17]. Thus, we propose a mechanism to faithfully replay past events under snapshot isolation with low recording overhead. $\mathtt{R}^3$ records coarse-grained per-transaction snapshot information during normal execution and reconstructs

equivalent snapshots during replay. Our evaluation shows that R^3 adds small performance and storage overhead during normal execution, making "always-on" recording practical.

The main challenge in retroaction is to maintain fidelity with the original trace of concurrent executions while allowing nearly arbitrary code modifications. For example, developers may want to test a bug fix for a race condition that only occurs when two concurrent requests are interleaved in a specific manner. However, this is challenging because code modifications may alter execution paths unpredictably, and there is no ground truth for events that did not originally occur. Thus, we define and provide high-fidelity retroaction which ensures that transactions maintain their original order and the concurrency of their schedules. We show this allows developers to fairly compare the behavior of a retroactive execution with that of the original execution. We further improve the performance of retroaction by developing an algorithm to selectively re-execute relevant requests based on their data dependencies.

Since R^3 only requires the DBMS to provide at least snapshot isolation, it supports most production DBMSs we consider without requiring any modifications to them. We implement R^3 using Postgres as the application DBMS backend. We demonstrate that R^3's powerful features can effectively and efficiently help debug real, hard-to-reproduce concurrency bugs from popular open-source web applications such as Moodle and WordPress. Our evaluation shows that R^3 always-on recording adds a small runtime performance overhead (up to 25% for point queries but <0.1% for more complex transactions like in TPC-C). Moreover, storage overhead is only 42 – 100 bytes per request, so if an application serves on average 1K requests/sec, R^3 can store more than three months of traces in a single 1TB disk drive and let developers replay and retroactively execute requests from any time in that trace.

In summary, our key contributions are:

(1) We show that replay and retroaction are uniquely practical for database-backed distributed applications because they access shared state at a coarse transaction-level granularity.
(2) We develop novel replay and retroaction mechanisms and prove strong correctness guarantees under snapshot isolation. We additionally introduce novel optimizations such as parallel and selective execution to further improve efficiency.
(3) We demonstrate that R^3 helps debug real, hard-to-reproduce concurrency bugs in open-source web applications. We evaluate R^3 using TPC-C and microservice workloads and show that its recording adds small runtime and storage overhead to normal execution, and it can faithfully replay past requests within 0.33 – 1.6× and retroactively execute them within 0.11 – 0.78× of the original request execution time.

2 R^3 OVERVIEW

R^3 integrates with application production and testing environments and DBMSs to provide efficient replay and retroaction without requiring modifications to application business logic or to the DBMS. We sketch R^3's architecture in Figure 1. It has three components:

- The **interceptor** traces application request execution and database transactions during normal execution in production and exports the information asynchronously to the *data recorder*.

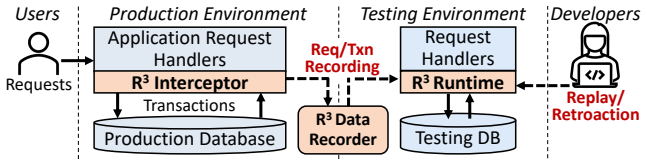

Figure 1: R^3 integrates with application production and testing environments. R^3's lightweight interceptor traces request and transaction information and sends it to the data recorder to enable replay and retroaction in the R^3 runtime.

- The **data recorder** stores recorded request and transaction information in an analytical database for replay and retroaction.
- The **runtime** controls and coordinates application request handler and transaction execution in the testing environment. This is the core component that performs the R^3 replay and retroaction algorithms. Developers interact with the runtime to replay or retroactively execute bugfixed code over recorded traces.

In the remainder of this section, we outline the requirements R^3 makes of applications and the DBMS to provide efficient replay and retroaction. We design R^3 to work with a wide range of applications and most production DBMSs, for example by supporting snapshot isolation. We additionally discuss R^3's limitations.

2.1 Requirements for Applications

Our design of R^3 targets database-backed distributed applications, such as an e-commerce microservice application or an online forum. We assume applications consist of many request handlers, and each handler may execute multiple transactions. For example, an online forum might have a "create post" handler and a "read post" handler. This is a common pattern for microservices and web service backends. Each request invokes its corresponding handler, which may in turn invoke other handlers (e.g., through RPCs). We assume each request is assigned a unique request ID (`reqId`) that is propagated through all handlers it invokes; this is a common practice.

To provide faithful replay and retroaction efficiently, R^3 requires applications to follow three principles:

- Store all application shared state in databases.
- Access or update shared state only through transactions.
- Be deterministic: a request's output and state changes must be determined only by its input and the database state it observes.

Request handlers can maintain local state for individual requests, but any state shared across requests must be stored in databases. R^3 does not rely on a specific data model; therefore, applications can choose to use relational DBMSs or non-relational but transactional stores like the high-performance key-value store FoundationDB which provides strictly serializable isolation.

These requirements are practical as they align with two important trends in distributed applications. First, developers increasingly embrace microservice architectures and deploy applications on serverless platforms (e.g., AWS Lambda [5]). Such applications naturally follow R^3 principles as they handle requests with workflows of stateless and deterministic functions and manage state using cloud databases. Second, many non-relational data stores are increasingly supporting transactions [43]. For example, the document

store MongoDB now supports snapshot isolation [20]. Therefore, it will only be easier for developers to follow R^3 principles.

For simplicity, we assume each request handler invocation is single-threaded (an application runs concurrent handlers in parallel threads), and each application uses a single database. R^3 can work for applications using multiple databases if their transaction logs are aligned (e.g., using a cross-database transaction manager [9]).

2.2 Requirements for DBMS

R^3 is designed to work with most production DBMSs without modifying them. We make two fundamental requirements that are met by most production DBMSs:

- Support creating and restoring from backups.
- Support at least snapshot isolation (SI), as defined by Adya [1] for a transaction T: T always reads data from a snapshot of committed data valid as of the time T started, and updates of other transactions active after T started are not visible to T; T can commit only if, at commit time, no other concurrent transactions have already written data that T intends to write.

For the correctness of replay, if the DBMS supports serializable isolation, it needs to provide the *serial order* of committed transactions. If the DBMS supports only SI, we require the following instead:

- The DBMS provides the logical transaction start order. For simplicity, we assume transaction IDs reflect this order.
- Overlapping transactions can only block due to write conflicts.

Additionally, for SI, we have a requirement for performance:

- The DBMS provides an efficient representation of the snapshot information for each transaction, which refers to the set of committed transactions that are visible to it.

In Section 5.1, we discuss ways to obtain this snapshot information if the DBMS does not provide it.

2.3 Non-Goals

R^3 can faithfully replay successful executions and many types of program failures, including the most difficult concurrency bugs. However, there are certain issues that R^3 cannot reproduce because it would require either detailed knowledge of the runtime environment or fine-grained timing information.

- Aborted transactions. R^3 replay skips aborted transactions. We discuss possible extensions to support them in Section 3.6.
- External service calls. We assume external service calls are idempotent. Otherwise, they cannot be deterministically reproduced without the collaboration of external services.
- Performance issues. R^3 can replay slow queries but does not guarantee the same performance as the original execution.
- Environmental issues. R^3 does not capture runtime settings or reproduce crashes due to environmental issues such as out-of-memory errors or network failures.

3 FAITHFUL REPLAY

R^3 faithful replay allows developers to re-execute any past application requests, guaranteeing that the replayed request returns the same output as the original and applies the same state changes to the database. To make this possible when the database uses snapshot isolation, we develop a novel algorithm for deciding how to re-execute transactions that originally executed concurrently. In this section, we describe our coarse-grained tracing, introduce our replay algorithm, prove its guarantees, and discuss optimizations.

3.1 Always-On Recording

R^3 records the following per-transaction and per-request information during normal application execution:

- For each request, R^3 records its unique request ID, its input, and the IDs of all its transactions.
- For each transaction, R^3 records its unique transaction ID and snapshot information.
- For each transaction, R^3 records whether it committed or aborted. If it aborted, R^3 records the error message.

R^3 obtains transaction IDs and snapshot information from the DBMS, and intercepts the application handlers to record request IDs and user inputs. R^3 organizes this information into tables and exports it asynchronously to the data recorder. In particular, R^3 creates a recorded input table per application:

```
RecordedInput (reqId, serializedInput)
```

`reqId`, the primary key, is the unique request ID. R^3 serializes each original request's input into binary format.

R^3 also creates a transaction log table per application:

```
TransactionLog (txnId, reqId, snapshot, status)
```

`txnId`, the primary key, is the unique transaction ID. `reqId` is a foreign key referencing the `RecordedInput` table.

Section 5.1 discusses how we capture this information and implement an efficient buffer to export it to an analytical database.

3.2 End-to-End Replay Workflow

To replay past requests, a developer specifies a range of request IDs. R^3 replays the specified requests following these steps:

(1) R^3 restores the database to the state it was in immediately before the first replayed request originally executed. We discuss our implementation of this in Section 5.2.
(2) Before replaying a request, R^3 retrieves its original user input from the data recorder and allocates a thread to execute its application code and transactions. R^3 also adds a breakpoint before each transaction in the request handler's code.
(3) The request threads execute their handlers' code, stopping at each breakpoint. A coordinator thread runs Algorithm 1 (discussed in Section 3.3), signaling the request threads when to execute and commit each transaction.

R^3 repeats step (2) and step (3) until all requests and transactions in the specified range finish.

3.3 Replay Algorithm

The main challenge in replay is supporting transactions that originally executed concurrently under snapshot isolation. To address this, we develop a novel algorithm (Algorithm 1). The high-level idea is that R^3 should execute each transaction over a snapshot that is equivalent to the one it saw in its original execution. R^3 executes transactions sequentially based on their original start order. However, it does not commit a transaction T right after it completes. This

Algorithm 1 R^3 Replay for Snapshot Isolation

```
1:  dr ← connectDataRecorder()              ▷ Connect to the data recorder.
2:  rt ← getRuntime()      ▷ Runtime for executing handlers/transactions.
3:  function REPLAY(beginReq, endReq)       ▷ Replay [beginReq, endReq)
4:      startOrder ← dr.getStartOrder(beginReq, endReq)
5:      for t ∈ startOrder do
6:          for s ∈ t.snapshot do           ▷ Commit transactions visible to t.
7:              await(s.execCompleted)      ▷ Wait for s execution to complete.
8:              rt.commit(s)                ▷ Signal runtime to commit s.
9:          if t.originallyCommitted then
10:             rt.execOnly(t)              ▷ Signal runtime to execute t (do not commit).
11:         else
12:             rt.returnError(t)           ▷ Signal runtime to skip t and return error.
13:     for t ∈ startOrder do               ▷ Commit remaining transactions.
14:         await(t.execCompleted)
15:         rt.commit(t)
```

ensures the next few transactions do not see the writes of T, but rather see their expected snapshot. T commits immediately before the execution of the first transaction that has T in its snapshot.

To perform replay following Algorithm 1, R^3 executes all requests and their transactions between `beginReq` (included) and `endReq` (excluded). R^3 first queries the data recorder to retrieve a list of all transactions in requests between [`beginReq`, `endReq`) in the order in which they started (line 4).

After retrieving transaction information, R^3 sequentially executes each transaction following this start order, using Algorithm 1 to coordinate the execution of request threads. When a request thread encounters a breakpoint (before each transaction), it waits for a signal from the coordinator thread that runs Algorithm 1 before executing the transaction. Request threads do not immediately commit transactions after they complete. Instead, before signaling the execution of a transaction, the coordinator checks if there are any transactions in its snapshot which have not committed. If there are, it signals their request threads to commit them (lines 6 – 8). After executing all transactions, the coordinator signals the request threads to commit all remaining uncommitted transactions (lines 13 – 15). Note that if a transaction originally aborted, R^3 does not execute it during replay, but instead directly returns the recorded error. As we prove in Section 3.4, this algorithm is guaranteed to faithfully replay all transactions and requests without blocking.

For simplicity, in this section we assume transactions execute sequentially. In practice, they can often execute concurrently. In Section 3.5, we discuss strategies for parallelizing transactions and show they provide equivalent guarantees as sequential execution.

3.4 Replay Guarantees

We now prove the correctness of R^3's replay algorithm under snapshot isolation. We first prove two lemmas, then use them to prove a correctness theorem.

Replay Lemma 1. *Each replayed transaction sees the same snapshot as its original execution.*

Proof. Specifically, T observes all transactions as it did originally, and no other transactions. The former is easy to show: because R^3 executes transactions based on their original start order, transactions committed before T are guaranteed to have completed before T starts. Then, before executing T, R^3 commits all completed but uncommitted transactions that are in T's snapshot.

We show the latter by contradiction: Algorithm 1 does not commit a transaction T_c until right before the execution of the first transaction T_i that has T_c in its snapshot, and since the snapshot of a transaction is the set of transactions which committed before it started, T_c must also be in the snapshot of all transactions that start after T_i. Thus, for there to exist a transaction T_c which commits before T executes but is not in its snapshot, there must be a prior transaction T_i which begins before T but has T_c in its snapshot. However, this is impossible: if T_c committed before T_i that started before T, then T_c must also be in the snapshot of T. □

Replay Lemma 2. *During replay, a transaction T that originally committed will always commit and never block.*

Proof. Specifically, we show that T can execute without blocking. We require (Section 2.2) that transactions under snapshot isolation can only block due to write conflicts, so we must show that T has no write conflicts with concurrent transactions that committed in the original execution. Under snapshot isolation, a transaction can only commit if it does not have write conflicts with transactions that committed before it but are not in its snapshot. We have already shown in Lemma 1 that all replayed transactions observe the same snapshots they did originally. Since transactions are deterministic, their write sets must remain the same. Thus, T cannot have any write conflicts with concurrent transactions that committed before it, or T would not have originally committed. Conversely, T cannot have any write conflicts with concurrent transactions that committed after it, or else they would not have originally committed. Therefore, T has no write conflicts with concurrent transactions and does not block during replay. If T committed originally, does not block during replay, observed the same snapshot as it did originally, and is deterministic, then it must commit during replay. □

Theorem 1. *When replaying a request, R^3 returns the same final output (or error state) as the original execution and applies the same state changes to the database.*

Proof. We require (Section 2.1) that request handlers are deterministic, so their output and state changes are determined entirely by their input and the database state they observe. R^3 supplies replayed requests with the same input they saw originally. Moreover, we proved in the two previous lemmas that each transaction in a replayed request observes the same snapshot of database state that it did originally, and that replayed requests never block. Therefore, requests must return the same final output (or error state) and apply the same state changes to the database. □

3.5 Optimizations

Algorithm 1 executes and commits transactions sequentially, however, we can parallelize some transactions during replay. For example, suppose transactions T_1, T_2, and T_3 originally executed and committed following the order $s(T_1)s(T_2)c(T_1)c(T_2)s(T_3)c(T_3)$. During replay, T_1 and T_2 can execute concurrently because T_1 is not in the snapshot of T_2. Similarly, before starting T_3, we can commit T_1 and T_2 in parallel. Moreover, if T_1 and T_2 are read-only, they can commit right after execution and T_3 does not need to wait for them as they

do not change the database. Since snapshot isolation guarantees that a transaction cannot see effects from concurrent transactions, parallel execution does not impact replay correctness. To speed up transaction commit during replay, we note that replayed transactions need not be durable, so if the database allows it, a commit can succeed without waiting for writes to be flushed to disk.

3.6 Discussion and Possible Extensions

Other Isolation Levels. R^3 can record and replay at transaction-level granularity because under SI, the behavior of committed transactions depends only on the order in which they start and commit. To support other isolation levels where the outcome of a query depends on the schedule of queries across concurrent transactions (e.g., isolation levels that allow anomalies such as phantoms or non-repeatable reads), we must record fine-grained timing information for each data operation and reconstruct the same schedule during replay. For instance, to faithfully replay under read committed (RC), we must record the snapshot and timing for each query and control the replay execution at query-level granularity.

Aborted Transactions. Aborts can be application-induced (e.g., `ABORT` issued by a handler) or conflict-induced (e.g., lock contentions and constraint violations). To replay a single application-induced aborted transaction without other transactions, R^3 can simply re-execute it and roll it back after the `ABORT`. However, faithfully replaying aborts in the presence of concurrent transactions requires knowledge of concurrency control implementation and detailed timing information for data operations, even under SI or serializable isolation. For instance, consider transactions T_1 and T_2 originally followed $s(T_1)s(T_2)\texttt{abort}(T_2)c(T_1)$, where T_2 was aborted by the application; T_2 also had a write conflict with T_1, but T_2 acquired the lock first and aborted before T_1 needed the lock. To replay them faithfully, we must ensure that T_2 acquires the lock first and aborts before T_1 needs the lock, so T_2 can abort due to the same application-induced abort rather than lock contention. Therefore, this coordination of individual data operations across transactions adds overhead for both recording and replay.

4 RETROACTION

In addition to faithful replay, R^3 also supports retroaction: developers can modify their code and re-execute it on traces of past requests. Retroaction is challenging because there is no ground truth for executions that never happened originally. Thus, we provide *high-fidelity* retroaction, defined as retroaction that preserves transaction order and the concurrency of their schedules, so developers can fairly compare a retroactive execution to the original one, for example to test bug fixes. In this section, we discuss what code modifications are supported by R^3, propose our retroaction algorithm, prove its guarantees, and introduce optimizations.

As a running example, we use a request handler (simplified) from the popular online education platform Moodle, as shown below.

```
def subscribeUser(userId, forumId):
  isSub = execTxn(isSubscribed(userId, forumId))
  if (not isSub):
    execTxn(forumInsert(userId, forumId))
```

This handler subscribes a user to a forum and contains a race condition (MDL-59854 [22]). Since the handler checks in one transaction if a user is subscribed to a forum and then in a separate transaction subscribes them to a forum, a user can be subscribed to the same forum multiple times if concurrent requests are interleaved.

4.1 Supported Modifications

To guarantee that R^3 can always re-execute those past requests, we assume the modified code is compatible with past inputs. Moreover, we do not allow modifications to the recorded traces other than skipping specific past requests; new requests cannot be added, and existing requests cannot be modified. R^3 requires that all modifications to application code follow the principles defined in Section 2.1. We categorize modifications as follows:

Retroactive Analysis. Developers may want to write analysis code inside the original handler code. R^3 supports arbitrary analysis code as long as it does not alter the handler's logic: it must run the same queries in the same transactions in the same order. If modifications only contain retroactive analysis, R^3 can reuse its replay algorithm to execute the modified code and faithfully reproduce past program state with the guarantees in Section 3.4. This is useful for debugging. For example, as shown below, developers can inspect a past execution by retroactively adding logging statements.

```
def subscribeUser(userId, forumId):
  isSub = execTxn(isSubscribed(userId, forumId))
  if (not isSub):
    LOG("Add User " + userId + " Forum" + forumId)
    execTxn(forumInsert(userId, forumId))
```

Retroactive Modification. Developers may want to change application code both inside transactions (e.g., changes to query parameters or query statements) and outside transactions (e.g., changes to transaction execution order within a request). Retroactive modification may not change the number of transactions to be executed given the same user request input. Retroactive modification is useful for verifying bug fixes and for testing new application features over past events. For example, as shown below, we could fix the Moodle duplication issue by changing the forum insert to an upsert (and adding a uniqueness constraint over forumId and userId pair), then test this fix using retroactive modification.

```
def subscribeUser(userId, forumId):
  isSub = execTxn(isSubscribed(userId, forumId))
  if (not isSub):
    execTxn(forumUpsert(userId, forumId))
```

Transaction Deletion. Developers may want to reduce the number of transactions for serving a request. For example, if a request handler originally contains two transactions, but the developer later decides to merge the second transaction into the first one, then the second transaction is effectively deleted. Wrapping multiple transactions into one transaction is a common strategy to fix bugs [32]. For example, as shown below, we could fix the Moodle duplication issue by performing both the check and insert in one transaction, then test this fix using retroactive deletion.

```
def subscribeUser(userId, forumId):
  beginTxn()
  if (not isSubscribed(userId, forumId)):
    forumInsert(userId, forumId)
  commitTxn()
```

Transaction Addition. Developers may also want to add additional transactions to a request, for example to split a large transaction into smaller transactions to improve application performance, or to introduce new functionality. Transaction addition increases the number of transactions to be executed given the same user request input. For example, as shown below, we can add a new transaction to fetch a list of all the forum's subscribers.

```
def subscribeUser(userId, forumId):
  isSub = execTxn(isSubscribed(userId, forumId))
  if (not isSub):
    execTxn(forumInsert(userId, forumId))
  subscribers = execTxn(fetchSubscribers(forumId))
```

4.2 Retroaction Goals

R^3 allows nearly arbitrary modifications to application code, which may alter execution paths unpredictably. For example, a transaction that committed originally may abort during retroaction due to new write conflicts, and a transaction that aborted originally may commit because a modification fixed the issue that caused the original abort. Therefore, unlike in replay, there exists no ground truth to faithfully reproduce. Our goal is to maintain *high fidelity* with the original trace of concurrent transactions. We define high-fidelity retroaction as executions that preserve transaction order and the concurrency of their schedules, providing these guarantees:

- Retroactive execution executes all requests without blocking.
- Concurrent transactions from parallel requests that committed originally are still concurrent.
- A transaction observes all transactions (if they commit) it did originally and possibly transactions that aborted originally, but it cannot see other originally committed transactions that were not in its original snapshot.

These guarantees allow developers to fairly compare the behavior of a retroactive execution to that of the original execution. For example, consider transactions T_1, T_2, and T_3 followed $s(T_1)s(T_2)c(T_1)c(T_2)s(T_3)c(T_3)$. R^3 ensures that retroaction preserves both the order (T_3 executes after T_1, T_2) and the concurrency of their schedules (T_2 does not see T_1). If the bug occurs only when T_1 and T_2 are concurrent and T_3 executes after them, R^3 allows developers to test their bug fix under the same conditions. For instance, the Moodle bug [22] only occurs when transactions from concurrent requests are interleaved. If R^3 were to execute without any coordination, this buggy scenario might never arise.

One important issue during retroaction is that sometimes, originally committed transactions are aborted due to write conflicts or other concurrency issues with new or modified transactions, then retried by request handler code. If this occurs, we treat the retry as a new transaction that may execute in a different order relative to other transactions, so the latter two guarantees do not apply to it.

4.3 Retroactive Execution

To retroactively execute modified code, developers must register all changes with R^3, then select a range of past requests for retroaction. Developers can also specify a list of requests to be skipped.

R^3 uses Algorithm 2 (we highlight major differences compared to Algorithm 1) to execute modified code over past requests while following the goals discussed in Section 4.2. Similar to replay, R^3

Algorithm 2 R^3 Retroactive Execution for Snapshot Isolation

```
1:  dr ← connectDataRecorder()          ▷ Connect to the data recorder.
2:  rt ← getRuntime()      ▷ Runtime for executing handlers/transactions.
3:  function RETROEXEC(beginReq, endReq, skipReqs)
4:      startOrder ← dr.getStartOrderDistinct(beginReq, endReq)
5:      for t ∈ startOrder do
6:          if t.reqId ∈ skipReqs ∨ rt.retroDeleted(t) then
7:              continue                          ▷ Skip this transaction.
8:          for s ∈ t.snapshot do   ▷ Commit non-skipped txns visible to t.
9:              await(s.execCompleted) ▷ Wait for s execution to complete.
10:             commitTxn(s)
11:         if t.originallyCommitted then
12:             rt.execOnly(t) ▷ Signal runtime to execute t (do not commit).
13:         else
14:             rt.execCommit(t) ▷ Signal runtime to execute and commit t.
15:         ▷ Note: Retroactively added transactions execute and commit
            immediately without requiring a signal.
16:     for t ∈ startOrder do          ▷ Commit remaining transactions.
17:         await(t.execCompleted)
18:         commitTxn(t)
19: function COMMITTXN(t)
20:     if ¬ t.aborted then
21:         rt.commit(t)                  ▷ Signal runtime to commit t.
22:     else
23:         rt.rollback(t)                ▷ Signal runtime to roll back t.
```

executes each request in a separate thread and stops before each transaction. A coordinator thread runs Algorithm 2, signaling the request threads when to execute and commit each transaction.

Note that Algorithm 2 retrieves a deduplicated list of transactions for the start order (line 4), omitting transactions that failed but retried in the original execution. R^3 executes transactions in their original start order and only tries to commit an originally committed transaction T immediately before the first transaction that has T in its snapshot. However, in retroaction, an originally committed transaction may abort. Thus, R^3 must check if a transaction has aborted before committing it (lines 19 – 23). If a transaction originally aborted but completes successfully, the request thread commits it immediately after it completes (line 14). If a handler has more transactions than it did originally (e.g., if a developer added a new one), the request thread executes and commits the new transactions without requiring a signal from the coordinator (line 15). If a transaction T is aborted and retried during retroactive execution, R^3 treats the retries as new transactions: the request thread retries T without requiring a signal from the coordinator.

R^3 supports transaction deletions and skipping requests. Therefore, R^3 checks if a transaction T must be skipped (lines 6 – 7), either because the developer specified to skip it or deleted it, or because it was omitted by our selective execution algorithm (Section 4.5).

One challenge in retroaction is that modifications may introduce write conflicts between transactions that did not previously conflict. For example, suppose transactions T_1, T_2, and T_3 originally executed and committed concurrently following the order $s(T_1)s(T_2)c(T_2)s(T_3)c(T_1)c(T_3)$, but a modification introduced a write conflict between T_1 and T_2. Algorithm 2 may hang indefinitely: T_2 requires a lock held by T_1, but T_1 will not commit until

T_2 commits and T_3 starts. To solve this problem, when a previously committed transaction T_i becomes blocked, $\mathtt{R}^3$ checks the DBMS to find which transaction T_j conflicts with it (e.g., by checking the lock holder from the DBMS). $\mathtt{R}^3$ aborts T_i if T_j is completed but pending commit, returning a database-specific conflict error to the request. In this example, $\mathtt{R}^3$ aborts T_2 so T_1 can proceed to commit.

4.4 Retroaction Correctness

We now prove that Algorithm 2 correctly provides the guarantees outlined in Section 4.2.

Retroaction Guarantee 1. *Retroactive execution executes all requests without blocking.*

Proof. If a transaction T is blocked during retroactive execution, $\mathtt{R}^3$ checks if it is blocked on a transaction that is completed but pending commit. If so, $\mathtt{R}^3$ aborts T (Section 4.3). Otherwise, $\mathtt{R}^3$ relies on the DBMS's deadlock detection to ensure that all transactions are eventually completed and either committed or aborted. □

Retroaction Guarantee 2. *Concurrent transactions from parallel requests that committed originally are still concurrent during retroaction (does not apply to retries).*

Proof. T_1 and T_2 are concurrent if neither is in the other's snapshot. Assuming T_1 started before T_2, they are concurrent if and only if T_1 committed after T_2 started. We will prove that their corresponding retroactive executions T_1' and T_2' remain concurrent. Retroactive execution preserves the start order of originally committed transactions, so if T_1 started before T_2, then T_1' starts before T_2'. Moreover, during retroactive execution, originally committed transaction T_1' does not commit until the start of the first transaction T_3' that has T_1' in its snapshot. T_3' must start after T_2', so T_1' must commit after T_2' starts and they remain concurrent. Note that this guarantee does not apply to transactions that, during retroactive execution, abort and are retried. For example, if T_1' aborts and is retried, the retried execution may have T_2' in its snapshot. □

Retroaction Guarantee 3. *A transaction observes all transactions (if they commit) it did originally and possibly transactions that aborted originally, but it cannot see other committed transactions that were not in its original snapshot (the latter does not apply to retries).*

Proof. For originally committed transactions, our retroactive execution algorithm is identical to our replay algorithm: transactions execute in their original start order and a transaction T does not commit until the start of the first transaction that had T in its original snapshot. Thus, this guarantee follows naturally from Replay Lemma 1: a retroactively executed transaction observes all originally committed transactions it originally observed (if they commit during retroactive execution without needing to be retried), but no other originally committed transactions. Similar to Retroaction Guarantee 2, the latter guarantee does not apply to retries, which may start later than their original execution and thus observe more transactions in their snapshot. □

Algorithm 3 $\mathtt{R}^3$ Selective Retroactive Execution of Request Types

```
1:  rd ← connectDataRecorder()
2:  rt ← getRuntime()
3:  function selectiveExecReqTypes(beginReq, endReq)
4:      allTypes = rd.getDistinctRequestTypes(beginReq, endReq)
5:      execTypes = rt.modified(allTypes) ▷ Initially only modified requests.
6:      wset = rt.getWriteTables(execTypes)                ▷ Write set.
7:      rset = rt.getReadTables(execTypes)                 ▷ Read set.
8:      reqs = rt.hasWrites(allTypes \ execTypes) ▷ Reqs containing writes.
9:      while ¬ reqs.isEmpty() do
10:         newReqTypes = {}
11:         for r ∈ reqs do
12:             if (r.writeSet ∩ (wset ∪ rset)) ∨ (r.readSet ∩ wset) then
13:                 ▷ Add request types with data dependencies on execTypes.
14:                 newReqTypes.add(r)
15:                 execTypes.add(r)
16:                 wset.add(r.writeSet)                   ▷ Update write set.
17:                 rset.add(r.readSet)                    ▷ Update read set.
18:         if newReqTypes.isEmpty() then
19:             break                  ▷ No more request types to be added.
20:         reqs.removeAll(newReqTypes)
21:     return execTypes
```

4.5 Selective Execution of Request Types

Retroactively executing every past request is expensive, especially when a modification is small and only affects a subset of past requests. We leverage data dependencies across transactions to selectively execute requests.

We sketch the selective execution algorithm in Algorithm 3. This is a static analysis algorithm that executes once before retroactive execution and returns a list of request types that must be re-executed. Invocations of the same request handler are considered to be the same request type. For example, users may send multiple requests to invoke the `subscribeUser` handler with different `userId` or `forumId` input parameters, but they belong to one request type (`subscribeUser`). The number of request types is typically much smaller than the total number of requests. This design guarantees that our algorithm scales to large traces.

The key idea is to always execute handlers that contain modified code, and only execute unmodified handlers if they have both data dependencies with retroactively executed transactions and contain updates to the application state. This algorithm is not guaranteed to skip all irrelevant requests, but ensures the re-execution of all requests that either serve as dependencies for modifications or are dependent on modified requests. In order to check dependencies, $\mathtt{R}^3$ assumes all queries are defined statically as parameterized prepared statements; this is a common practice to prevent SQL injection.

Algorithm 3 computes a transitive closure of request types (line 4) that must be re-executed within the range of `[beginReq, endReq)`. Specifically, we re-execute requests if their write sets intersect with the read or write sets of re-executed requests, or if their read sets intersect with the write sets of re-executed requests (line 12). We compute this closure statically and use table-level read and write sets, because computing it dynamically may require backtracking. For example, if the re-execution so far has a read and write set of Table {`A`}, then we are only re-executing transactions that write to Table `A`. If we then encountered a transaction that

reads from Table B then writes to Table A, we would be missing all writes to Table B, so we would have to restart from the beginning re-executing all transactions that write to Tables A or B. Given that we compute the closure statically, we must use table-level read and write sets because only table-level access information is known ahead of time for parameterized prepared statements; finer-grained dependencies may be path-dependent.

Supporting Data Correction. We can extend R^3 selective execution to efficiently correct past transactions. For example, in an online shopping web backend, developers might want to double the price of an item and re-calculate shipping fees for past orders. Developers may use R^3 to selectively re-execute relevant past order requests with the new price and update the production database to reflect the corrected fees. However, supporting data correction may require human intervention because updating past data might cause cascading effects on later requests. For example, customers might not have bought the item if the price were doubled. In this case, we must capture the causal relationships between requests and incorporate causality in our dependency checks.

5 IMPLEMENTATION

We implemented R^3 record, replay, and retroaction using Postgres as the application DBMS backend and Vertica as the analytical database for the data recorder. In this section, we discuss how our implementation records information on transactions and requests, efficiently exports that information to the data recorder, and restores the database to a past state for replay and retroaction.

5.1 Implementing R^3 Recording

Capturing Requests and Transactions. R^3 must capture four pieces of information for each request: its unique request ID, its input, the ID of each of its transactions, and the snapshot of each transaction (only required for SI). Obtaining the first three is straightforward, but capturing transaction snapshots depends on the DBMS.

If the DBMS exposes an efficient representation of transaction snapshot, we can directly use it. For example, our implementation leverages the Postgres `pg_current_snapshot()` system information function that returns the snapshot in a summary format with three fields: `xmin`, `xmax`, and `xip_list`. `xmin` is the smallest active transaction ID, `xmax` is one past the highest completed transaction ID, and `xip_list` is the list of active transactions at the time the snapshot was taken (under SI, it is the start of a transaction). During replay and retroaction, to check if a transaction T is in the current transaction's original snapshot, R^3 uses the following expression:

$$(T < \texttt{xmax}) \wedge (T \notin \texttt{xip\_list})$$

If the DBMS provides a globally logical order for transaction starts and commits (e.g., SQL Server records transaction begin and commit timestamps), R^3 can obtain the start and commit logical order/timestamp per transaction T at runtime. During replay, R^3 can compare the begin timestamp of T to the commit timestamps of previous transactions to decide which ones are visible to T.

If the DBMS supports SI but does not expose any information other than the start order (transaction IDs), we can maintain a `committed_txns` table that stores the list of committed transaction IDs. Transactions that contain writes must insert their transaction IDs into this table before they commit. Then, when a transaction T begins, R^3 can query this table and record the list of previous transactions that are visible to T. This alternative method may be further optimized for efficiency, for example, we can regularly delete old IDs to cap the number of rows in this table, but this is out of the scope of this paper and we leave it to future work.

Exporting Recorded Information. To minimize recording overhead, R^3 exports information to a remote data recorder asynchronously and in batches. R^3 maintains an in-memory buffer in the interceptor and appends recorded information to this buffer when processing requests. Periodically (in our implementation, every two seconds), a background thread exports the entire contents of the buffer to the data recorder (Vertica). Because captured information is buffered in memory and exported asynchronously, it is possible for it to be lost if the application server crashes before the buffer is exported, meaning requests that happened immediately before the crash could not be replayed. If developers cannot tolerate this data loss, we can optionally place the buffer on disk to guarantee its durability at some performance cost. We leave the decision of data retention policies to the developer; most analytical databases (including Vertica) have robust capabilities in this area.

5.2 Restoring Database State

R^3 replay and retroaction both require restoring the database to the state it was in immediately before the first request to be re-executed. Our Postgres-based implementation supports two database restoration methods. If the Postgres server is configured with write-ahead log (WAL) archiving, R^3 restores it to the latest backup and leverage Postgres's native support for point-in-time recovery to recover the database to the original transaction ID of the first transaction to be re-executed. This requires the application to have the full WAL archives since the backup. If WAL archives are unavailable, R^3 restores the server to the latest DBMS backup, then uses the replay algorithm to re-execute all transactions containing writes from the original backup time to the first transaction to be re-executed, thus faithfully recovering the database to the correct state.

6 CASE STUDIES

To concretely show how developers can use R^3 to debug and test their applications, we study common hard-to-reproduce concurrency bugs discussed in prior work on non-reproducible bugs [11] and server-side request races [32]. We examine three bugs from two popular open-source database-backed web applications, each with millions of users: the education platform Moodle (MDL) [23] and the content management system WordPress (WP) [40]. The bugs we study have different effects on the application (silent data corruption or database errors), different root causes (concurrent requests to the same or different handlers), and must be fixed in different ways (merge multiple transactions or modify queries). Each bug took developers substantial effort to reproduce and fix.

6.1 Moodle: Duplicate Entry

MDL-59854 [22] is a bug where concurrent requests to the same handler can cause silent data corruption. It is the issue on which we based the buggy Moodle forum subscription example in Section 4:

if two forum subscription requests are interleaved, a user may be subscribed to the same forum twice. Duplication occurred rarely in production because it required a user to send two identical requests simultaneously to the application. Thus, it was difficult to reproduce when it did occur: it took three months for developers debug and release the bug fix. The developer who reported this bug commented: *"you have to be pretty fast and pretty lucky to actually reproduce this issue."* By contrast, if Moodle is integrated with R^3, developers can easily reproduce the race condition by using R^3 to replay recorded past requests that contain the issue.

To fix the bug, developers initially attempted to merge the two transactions in the buggy request handler into one. When testing this bug fix retroactively using R^3, we found the bug may still occur under SI due to write skew: two concurrent transactions see the same snapshot and both add a subscription to the table. This issue was also observed by the Moodle developers, who eventually fixed it by adding a uniqueness constraint over the `(forumId, userId)` pair. Thus, R^3 retroaction enables developers to efficiently test whether a bug fix works.

6.2 WP-Option: Unique Constraint Violation

WP-11437 [39] is a bug where concurrent requests to the same handler may violate database constraints and cause errors. The buggy code contains two transactions that are similar to the previously-discussed Moodle forum subscription, which has a race condition when inserting a new option to the `option` table. Unlike in Moodle, the WordPress `option` table uses option name as a primary key, so the database returns a "duplicate key violates unique constraint" exception if multiple requests try to insert the same option.

To fix the bug, developers used `ON DUPLICATE KEY UPDATE` for the insert statement, which effectively turns the insert operation to an upsert. We test this bug fix using R^3 and find it eliminates constraint violation errors when retroactively executing a recorded trace with the fixed code. We still see serialization errors when multiple queries try to update the same option, but we fix this easily by retrying transactions that fail due to serialization errors. R^3 retroaction enables developers to effectively test whether a bug fix works and further improve the robustness of application code through better error handling.

6.3 WP-Comment: Inconsistent Status

WP-11073 [38] is a bug where concurrent requests to different handlers cause silent data corruption. Specifically, there is a race condition between adding a comment for a post (`AddComment`) and deleting the post and its comments (`TrashPost`). While deleting a post, the request handler first backs up the comment IDs and statuses to a `post_meta` table, and then in a separate transaction it updates all comment statuses to `trashed`. If `AddComment` executes in between these two steps, the new comment is not backed up in the `post_meta` table in the first transaction, but is still marked as trashed by the second transaction. This becomes a problem if a later request restores the deleted post and comments (`UntrashPost`). This restores all backed-up comments, but if new comments were not backed up, they are not restored.

To solve this issue, the developers added a query in `AddComment` to first check in the `post` table if a post is being deleted. If so, the comment fails and is never made visible to other users. We test this bug fix using R^3, finding that similarly to the Moodle issue, it only works if transactions run under serializable isolation. Otherwise, a write skew issue can occur where both `TrashPost` and `AddComment` see the same snapshot, so they execute concurrently without being aware of each other.

7 EVALUATION

We evaluate R^3 with workloads adapted from popular benchmarks and database-backed applications. We analyze the runtime and storage overhead of R^3 recording compared to a baseline that does not record or capture any application information. We also evaluate R^3 replay and retroactive execution performance. We show that:

(1) R^3 adds runtime overhead of <25% for point queries and <0.1% for complex transactions such as those in TPC-C. R^3 storage overhead is on average 42 – 100 bytes per request.
(2) R^3 can faithfully replay past recorded traces within 0.33 – 1.6× of the original execution time.
(3) R^3 can retroactively execute bugfixed code over recorded traces within 0.11 – 0.78× of the original execution time, by selectively re-executing only requests with data dependencies.

7.1 Experimental Setup

We implement R^3 in ~500 lines of Java code for recording transaction information, and ~1.2K lines of Java code for replay and retroactive execution. For our experiments, we use Postgres [29] v14.5 for application data and store R^3 recorded data in Vertica [37] v12.0.3. We use JDBC to communicate with Postgres and Vertica, and set Postgres connections to use the *Repeatable Read* (implemented as snapshot isolation) isolation level. For communications between request handlers, we use JeroMQ [30] v0.5.2 over TCP.

We run experiments on Google Cloud using `c2-standard-8` VM instances with 8 vCPUs, 32GB DRAM, and a SCSI HDD. We run the Postgres server, Vertica server, and application request handlers on separate VMs. We configure each application to use 128 parallel JDBC connections to communicate with Postgres. During replay and retroactive execution, we disable Postgres synchronous commit [28], as discussed in Section 3.5.

Baselines. To assess the overhead of R^3 recording during normal application execution, we use a *no-record* baseline that executes the benchmark workloads but does not retrieve snapshot information from Postgres and does not record per-request input. We run this baseline in a setup identical to that of R^3, but do not capture any transaction or request information or export any data to Vertica. To evaluate the effectiveness of our optimizations (Section 3.5) for replay, we use a *sequential* baseline that follows Algorithm 3.3 but executes and commits transactions sequentially.

7.2 Experimental Workloads

We evaluate R^3 using TPC-C as well as Moodle and WordPress from our case studies (Section 6). We implement these workloads in Java, use Postgres as the backend database, and follow their original table schema. To demonstrate that R^3 works for distributed applications, we adapt Moodle and WordPress using a microservice architecture: if a request contains multiple transactions, we implement each one

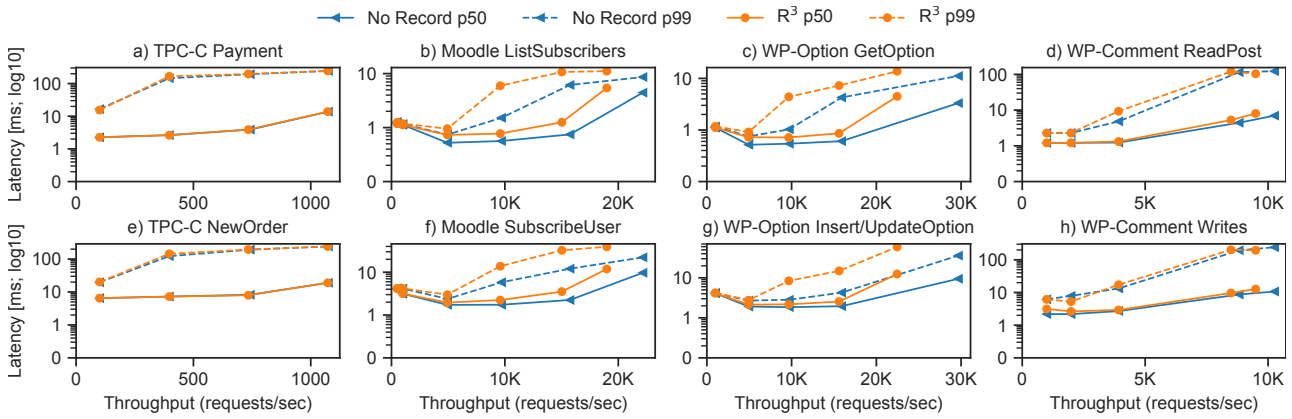

Figure 2: Throughput versus p50 and p99 latencies of R³ and a no-record baseline on TPC-C, Moodle, and WordPress workloads.

Workload	Operation	Ratio	Read-Only?	# of Txns.	Avg. # of SQL	Avg. Access State Size
TPC-C	Payment	50%	No	1	8	~500B
	NewOrder	50%	No	1	25	~2KB
Moodle	ListSubscribers	90%	Yes	1	1	~500B
	SubscribeUser	10%	No	2	2	~10B
WP-Option	GetOption	99%	Yes	1	1	~100B
	InsertOption	0.5%	No	2	2	~100B
	UpdateOption	0.5%	No	2	2	~100B
WP-Comment	ReadPost	80%	Yes	1	2	~10KB
	AddComment	10%	No	1	2	~1KB
	TrashPost	5%	No	2	5	~200B
	UntrashPost	5%	No	1	13	~200B

Table 1: Experimental workloads information. The last column shows the estimated state size that a request accesses.

in a separate RPC handler and compose handlers into a workflow to serve the request. As shown in Table 1, our workloads cover different scenarios for database-backed applications.

TPC-C. We first benchmark R³ with the `Payment` and `NewOrder` transactions from TPC-C, the industry-standard benchmark for OLTP databases. The NewOrder transaction mimics customers submitting orders to their local warehouse district. The Payment transaction mimics customers making payments on the submitted orders. We choose these two transactions because they comprise 90% of the TPC-C workload. Our workload consists of 50% `Payment` and 50% `NewOrder`. We populate the TPC-C database with 24 warehouses.

Moodle. Our Moodle workload consists of 10% requests subscribing users to forums (`SubscribeUser`) and 90% requests retrieving the subscribers to a forum (`ListSubscribers`). We pre-load 1000 forums, and for each forum we initially load one subscriber. Note that the `SubscribeUser` handler contains the race condition described in Section 6, which may create duplicate forum subscriptions.

WP-Option. Our first WordPress workload consists of 99% requests retrieving an option value (`GetOption`) and 1% requests setting an option value, split evenly between inserts (`InsertOption`) and updates (`UpdateOption`). We pre-load 10K options with a 10B option key and 100B option value. Note that the `InsertOption` handler contains the bug discussed in Section 6, which may cause primary key errors.

WP-Comment. Our second WordPress workload consists of 80% requests reading a post (`ReadPost`), 10% requests adding comments on a post (`AddComment`), 5% requests deleting a post and its associated comments (`TrashPost`), and 5% requests undoing a delete (`UntrashPost`). We pre-load 2000 posts, each starting with 10 comments (for an initial total of 20K comments). Each post and comment contains 1KB of text. Note that the `AddComment` handler contains the bug discussed in Section 6, which may cause inconsistent comment statuses after undoing a post deletion.

7.3 Recording Overhead Analysis

Runtime Overhead Analysis. We first investigate R³ recording overhead. We execute all four workloads with R³ recording enabled and compare performance to the no-record baseline, showing results in Figure 2. We find that R³ shows negligible (<0.1%) performance difference for TPC-C and adds throughput and latency overhead of <10% for the WP-Comment workload, but overhead increases to 25% for WP-Option and Moodle.

To further investigate the causes of R³ recording overhead, we show in Figure 3 the latency breakdown of individual point read and point write operations with R³ recording enabled. We find that most R³ overhead comes from the implementation of Postgres: to retrieve snapshot information, each transaction must make a `pg_current_snapshot` query, which adds an additional round trip to the database. This fixed per-transaction overhead is significant for small transactions such as these point operations, but small for large transactions such as those in TPC-C and WP-Comment. We could further optimize this by modifying Postgres to return the snapshot in the same operation that initializes a transaction.

Storage Overhead Analysis. Our implementation of R³ exports recorded data to the column-oriented analytical database Vertica, which stores data in a compressed format while providing high performance for analytical queries. We find that the storage overhead for storing transaction information in Vertica is on average 24B per request. The overhead for storing recorded request input is application dependent. For TPC-C, storing input in Vertica requires

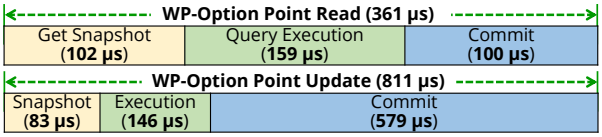

Figure 3: Latency breakdowns of the WP-Option point read and update transactions with R^3 recording. Each transaction needs a round trip to Postgres to retrieve snapshot.

76B per request because the original input contains a long list of integers such as customer IDs, warehouse IDs, and timestamps. For Moodle, input storage requires 18B per request because the original request input only contains a few integers representing user IDs and forum IDs. For WordPress, input storage requires 73B per request because the original request input contains 100B – 1KB of text. Thus, even in the worst case, an application would have to execute 10B requests to fill a 1TB disk drive with recording information.

7.4 Replay Performance

We next analyze the performance of R^3 replay. We combine the WordPress workloads, running a mix of 50% requests from WP-Comment and 50% from WP-Option. First, we collect data for replay by running TPC-C, Moodle, and the combined WordPress workloads for 60 seconds at different request rates. Then, we replay three workloads using R^3, with and without our optimizations that execute and commit transactions in parallel, and measure the total execution time. We show results in Figure 4.

We find that our optimizations improve performance by up to 7.2× compared to the sequential baseline, because we leverage parallelisms across concurrent transactions. Note that the improvement is greater at high request rates because the original trace would have higher concurrency and can be parallelized across more threads.

For optimized parallel executions, R^3 can always replay TPC-C workloads within 60 seconds. For Moodle and WordPress, R^3 parallel replay is slower than the original execution at high load because it must follow the original execution's start order and snapshot information. Specifically, R^3 must validate that each replayed transaction T observes the correct snapshot, committing all uncommitted transactions in T's snapshot if necessary. This is not a problem for complex transactions like in TPC-C where threads are mainly occupied by transaction execution and commit; R^3 can achieve better throughput than the original execution because it uses non-durable commits. However, for small, read-mostly transactions like in Moodle and WordPress, R^3 coordination may cause longer stalls and not saturate threads as in their original executions.

7.5 Retroactive Execution Performance

We now analyze the performance of R^3 retroaction with Moodle and WordPress. We re-execute collected traces from Section 7.4 with the bug fixes discussed in Section 6. We perform re-execution with and without the selective execution optimization, which uses a static analysis algorithm to only re-execute requests that have data dependencies with modified requests. Both use parallel executions as discussed in Section 3.5. We show results in Figure 5.

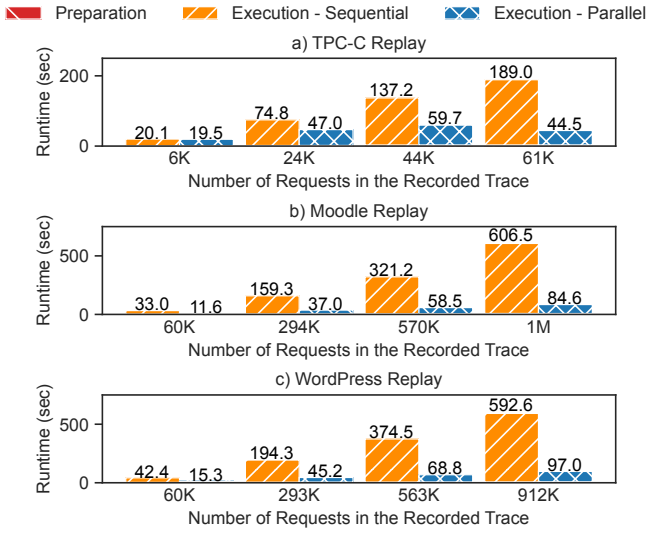

Figure 4: Execution time for replaying a trace that originally executed in 60 seconds, varying the number of requests in (thus the throughput of) the original execution. We show results with and without parallel execution optimizations. We break down runtime into preparation (retrieving recorded information) time and replay execution time.

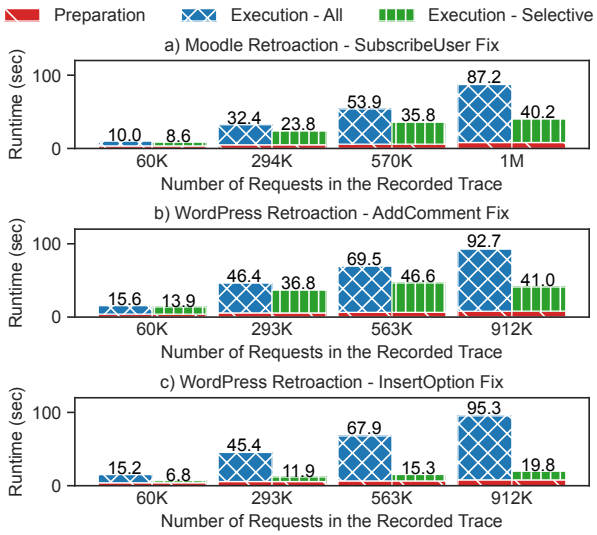

Figure 5: Execution time for retroactively executing bugfixed code over a trace that originally executed in 60 seconds, varying the number of requests in (thus the throughput of) the original execution. We show results with and without the selective execution optimization. We break down runtime into preparation (retrieving recorded information) time and retroactive execution time.

Without the selective execution optimization, retroaction takes up to 1.6× longer than the original trace execution, similar to replay

and for the same reasons. However, our selective execution optimization decreases re-execution time by 1.16 – 4.8× by skipping transactions unrelated to the modified request, so the optimized retroaction takes less time than the original execution.

8 RELATED WORK

Record and Replay. There has been much prior work on deterministic record and replay for both operating system kernels and user-level programs. Whole-system replay tools like Arnold [8] and OmniTable [33] require expensive and heavyweight instrumentation (e.g., kernel modification) to capture detailed timing of individual instructions. SMT-ReVirt [10] replays the entire VM and all applications by modifying the hypervisor and using hardware page protection to accurately capture accesses to shared state, which incurs high runtime overhead. RR [27] supports record and replay for unmodified user-level applications running with stock Linux kernels. However, RR is limited to single-threaded execution and causes a large slowdown for programs with high parallelism. R2 [14] can efficiently record and replay multi-threaded applications but may replay incorrectly if the program has race conditions. REPT [7] and Kernel REPT [13] use a circular buffer to record detailed information on the last few instructions to execute on each thread, which enables reproducing recent system failures but does not allow replaying arbitrary past executions.

Several record and replay systems rely on specialized hardware or propose new hardware architectures. For example, BugNet [24] and FDR [42] propose new hardware architectures to continuously trace program execution and provide enough information to deterministically replay the last several instructions before a system crash. Castor [18] leverages hardware transactional memory to support low-overhead always-on record and replay for multi-core applications, but it cannot deterministically replay all data races. DeLorean [21] proposes a new hardware architecture where processors execute atomic blocks of instructions, so it only needs to record the commit order of these blocks for faithful replay.

Compared to existing tools, $\mathtt{R}^3$ focuses on database-backed distributed applications and achieves low-overhead always-on recording by capturing traces at transaction-level granularity. $\mathtt{R}^3$ does not require modifications to applications, the OS kernel, or hardware.

Transaction Reenactment. Reenactment [2–4, 26] is a technique that replays and retroactively captures data provenance of past transactions running under SI or read committed in MVCC DBMSs. It provides detailed information about how tuples were derived through past updates, but unlike $\mathtt{R}^3$, it works only for collections of SQL statements and does not support procedural code such as application business logic. It relies on DBMS features such as time-travel queries and audit logging, resulting in non-trivial overhead from fine-grained tracking [2]. In comparison, $\mathtt{R}^3$ focuses on application-level time travel: it supports end-to-end replay and retroaction for entire application request handlers and makes this practical by providing always-on low-overhead recording in production.

Retroactive Program Execution. Existing systems that support retroactive execution either limit code modifications or have restrictive programming semantics, and they do not support concurrent executions. Retroactive Aspects [35] is designed to analyze and replay Linux kernel execution. Its implementation only allows instrumenting the Linux kernel, and while it supports evaluating analysis code (e.g., logging) in the past, it does not allow retroactive modification of kernel logic. The GProM debugger [26] uses reenactment to test changes to SQL queries in one transaction at a time; their following work Mahif [6] can efficiently test hypothetical changes to multiple transactions, but unlike $\mathtt{R}^3$, it only considers a serial history, not a concurrent one, and does not support procedural code. Reverb [25] supports speculative bug fix analysis, which is similar to $\mathtt{R}^3$ retroaction. Reverb replays a past application execution to a point and allows developers to edit the code or data of the application; post-edit, Reverb follows the recorded event order and other non-determinisms in the trace. However, unlike $\mathtt{R}^3$, Reverb requires both the server and client to be single-threaded and event-driven. $\mathtt{R}^3$ retroaction is inspired by Retro-λ [19], but that system only supports retroaction for event-sourced serverless applications built with the command query responsibility segregation (CQRS) architectural pattern. Retro-λ also only considers a single-threaded isolated microservice and does not support transactions.

Deterministic Databases. $\mathtt{R}^3$ faithful replay is related to prior work on deterministic databases [12, 16, 31, 34, 36, 44]. These guarantee serializable transactions by predetermining a serial order prior to execution through a sequencing layer. They usually run transactions in batches to reduce the sequencing overhead. They also support efficient replication without coordination by replaying the same batches of transactions with the deterministic serial order. Whether the DBMS is deterministic is orthogonal to $\mathtt{R}^3$ design. If the application uses a deterministic database, $\mathtt{R}^3$ only needs to generate one record per transaction batch and can faithfully replay past executions by sequentially running recorded batches.

Database Command Logging. Command logging is a recovery scheme for main-memory databases [17, 41, 44], where instead of logging modifications to data, the DBMS only records the transaction's logic (e.g., SQL queries). Then, to recover, the DBMS starts from a checkpoint and replays the commands in the log. Command logging significantly reduces log data sizes because, similar to $\mathtt{R}^3$, it only needs one record per transaction. However, it only supports serializable isolation and requires the commit order to be the same as the serial order. Otherwise, sequentially replayed transactions running with non-serializable isolation levels (like snapshot isolation) may cause divergence [17] (we also discuss this issue in Section 3). $\mathtt{R}^3$ solves this issue by proposing a novel algorithm to efficiently replay transactions executed with snapshot isolation.

9 CONCLUSION

We presented $\mathtt{R}^3$, a tool that can faithfully replay recorded past executions and retroactively execute modified code over past requests for database-backed distributed applications. $\mathtt{R}^3$ only requires the database to support at least snapshot isolation, and implements a lightweight interceptor to record concurrency information for distributed applications at a coarse transaction-level granularity. We evaluate $\mathtt{R}^3$ on popular open-source applications and show its always-on recording has small runtime overhead during normal application execution and can help debug real, hard-to-reproduce concurrency bugs.

REFERENCES

[1] Atul Adya. 1999. *Weak consistency: a generalized theory and optimistic implementations for distributed transactions.* Ph.D. Dissertation. Massachusetts Institute of Technology, Dept. of Electrical Engineering and Computer Science.

[2] Bahareh Arab, Dieter Gawlick, Vasudha Krishnaswamy, Venkatesh Radhakrishnan, and Boris Glavic. 2016. *Formal foundations of reenactment and transaction provenance.* Technical Report. Technical report, IIT.

[3] Bahareh Sadat Arab, Dieter Gawlick, Vasudha Krishnaswamy, Venkatesh Radhakrishnan, and Boris Glavic. 2016. Reenactment for Read-Committed Snapshot Isolation. In *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management* (Indianapolis, Indiana, USA) *(CIKM '16)*. Association for Computing Machinery, New York, NY, USA, 841–850. https://doi.org/10.1145/2983323.2983825

[4] Bahareh Sadat Arab, Dieter Gawlick, Vasudha Krishnaswamy, Venkatesh Radhakrishnan, and Boris Glavic. 2017. Using reenactment to retroactively capture provenance for transactions. *IEEE Transactions on Knowledge and Data Engineering* 30, 3 (2017), 599–612.

[5] AWS. 2022. AWS Lambda. https://aws.amazon.com/lambda/.

[6] Felix S. Campbell, Bahareh Sadat Arab, and Boris Glavic. 2022. Efficient Answering of Historical What-If Queries. In *Proceedings of the 2022 International Conference on Management of Data* (Philadelphia, PA, USA) *(SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 1556–1569. https://doi.org/10.1145/3514221.3526138

[7] Weidong Cui, Xinyang Ge, Baris Kasikci, Ben Niu, Upamanyu Sharma, Ruoyu Wang, and Insu Yun. 2018. {REPT}: Reverse debugging of failures in deployed software. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*. 17–32.

[8] David Devecsery, Michael Chow, Xianzheng Dou, Jason Flinn, and Peter M. Chen. 2014. Eidetic Systems. In *OSDI 2014*. Broomfield, CO, 525–540.

[9] Akon Dey, Alan Fekete, and Uwe Röhm. 2015. Scalable distributed transactions across heterogeneous stores. In *ICDE 2015*. 125–136.

[10] George W Dunlap, Dominic G Lucchetti, Michael A Fetterman, and Peter M Chen. 2008. Execution replay of multiprocessor virtual machines. In *Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*. 121–130.

[11] Mona Erfani Joorabchi, Mehdi Mirzaaghaei, and Ali Mesbah. 2014. Works for me! characterizing non-reproducible bug reports. In *MSR 2014*. 62–71.

[12] Jose M Faleiro, Daniel J Abadi, and Joseph M Hellerstein. 2017. High performance transactions via early write visibility. *Proceedings of the VLDB Endowment* 10, 5 (2017).

[13] Xinyang Ge, Ben Niu, and Weidong Cui. 2020. Reverse debugging of kernel failures in deployed systems. In *Proceedings of the 2020 USENIX Conference on Usenix Annual Technical Conference*. 281–292.

[14] Zhenyu Guo, Xi Wang, Jian Tang, Xuezheng Liu, Zhilei Xu, Ming Wu, M Frans Kaashoek, and Zheng Zhang. 2008. R2: An Application-Level Kernel for Record and Replay.. In *OSDI*, Vol. 8. 193–208.

[15] Qian Li, Peter Kraft, Michael Cafarella, Çağatay Demiralp, Goetz Graefe, Christos Kozyrakis, Michael Stonebraker, Lalith Suresh, and Matei Zaharia. 2023. Transactions Make Debugging Easy. In *CIDR 2023*.

[16] Yi Lu, Xiangyao Yu, Lei Cao, and Samuel Madden. 2020. Aria: a fast and practical deterministic OLTP database. *Proceedings of the VLDB Endowment* 13, 12 (2020), 2047–2060.

[17] Nirmesh Malviya, Ariel Weisberg, Samuel Madden, and Michael Stonebraker. 2014. Rethinking main memory OLTP recovery. In *2014 IEEE 30th International Conference on Data Engineering*. IEEE, 604–615.

[18] Ali José Mashtizadeh, Tal Garfinkel, David Terei, David Mazieres, and Mendel Rosenblum. 2017. Towards practical default-on multi-core record/replay. *ACM SIGPLAN Notices* 52, 4 (2017), 693–708.

[19] Dominik Meissner, Benjamin Erb, Frank Kargl, and Matthias Tichy. 2018. Retro-λ: An event-sourced platform for serverless applications with retroactive computing support. In *DEBS 2018*. 76–87.

[20] MongoDB. 2023. MongoDB Transactions. https://www.mongodb.com/docs/manual/core/transactions/.

[21] Pablo Montesinos, Luis Ceze, and Josep Torrellas. 2008. Delorean: Recording and deterministically replaying shared-memory multiprocessor execution efficiently. *ACM SIGARCH Computer Architecture News* 36, 3 (2008), 289–300.

[22] Moodle. 2017. Duplicate forum subscriptions due to race conditions. https://tracker.moodle.org/browse/MDL-59854.

[23] Moodle. 2023. Moodle. https://moodle.org/.

[24] Satish Narayanasamy, Gilles Pokam, and Brad Calder. 2005. Bugnet: Continuously recording program execution for deterministic replay debugging. In *32nd International Symposium on Computer Architecture (ISCA'05)*. IEEE, 284–295.

[25] Ravi Netravali and James Mickens. 2019. Reverb: Speculative debugging for web applications. In *Proceedings of the ACM Symposium on Cloud Computing*. 428–440.

[26] Xing Niu, Boris Glavic, Seokki Lee, Bahareh Arab, Dieter Gawlick, Zhen Hua Liu, Vasudha Krishnaswamy, Su Feng, and Xun Zou. 2017. Debugging Transactions and Tracking their Provenance with Reenactment. *Proceedings of the VLDB Endowment (Demonstration Track)* 10, 12 (2017), 1857–1860.

[27] Robert O'Callahan, Chris Jones, Nathan Froyd, Kyle Huey, Albert Noll, and Nimrod Partush. 2017. Engineering Record and Replay for Deployability.. In *USENIX Annual Technical Conference*. 377–389.

[28] PostgreSQL. 2023. Asynchronous Commit. https://www.postgresql.org/docs/current/wal-async-commit.html.

[29] PostgreSQL. 2023. PostgreSQL. https://www.postgresql.org/.

[30] The ZeroMQ project. 2023. JeroMQ, Pure Java implementation of libzmq. https://github.com/zeromq/jeromq.

[31] Thamir M Qadah and Mohammad Sadoghi. 2018. Quecc: A queue-oriented, control-free concurrency architecture. In *Proceedings of the 19th International Middleware Conference*. 13–25.

[32] Zhengyi Qiu, Shudi Shao, Qi Zhao, Hassan Ali Khan, Xinning Hui, and Guoliang Jin. 2022. A Deep Study of the Effects and Fixes of Server-Side Request Races in Web Applications. In *MSR 2022*. 744–756.

[33] Andrew Quinn, Jason Flinn, Michael Cafarella, and Baris Kasikci. 2022. Debugging the OmniTable Way. In *OSDI 2022*. Carlsbad, CA, 357–373.

[34] Kun Ren, Dennis Li, and Daniel J Abadi. 2019. Slog: Serializable, low-latency, geo-replicated transactions. *Proceedings of the VLDB Endowment* 12, 11 (2019).

[35] Robin Salkeld, Wenhao Xu, Brendan Cully, Geoffrey Lefebvre, Andrew Warfield, and Gregor Kiczales. 2011. Retroactive aspects: programming in the past. In *Proceedings of the Ninth International Workshop on Dynamic Analysis*. 29–34.

[36] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J Abadi. 2012. Calvin: fast distributed transactions for partitioned database systems. In *Proceedings of the 2012 ACM SIGMOD international conference on management of data*. 1–12.

[37] Vertica. 2023. Vertica. https://www.vertica.com/.

[38] WordPress. 2009. Comment Status for Posts in the Trash. https://core.trac.wordpress.org/ticket/11073.

[39] WordPress. 2009. Option inserts triggered from front page can cause duplicate entry errors. https://core.trac.wordpress.org/ticket/11437.

[40] WordPress. 2023. WordPress. https://wordpress.com/.

[41] Yu Xia, Xiangyao Yu, Andrew Pavlo, and Srinivas Devadas. 2020. Taurus: lightweight parallel logging for in-memory database management systems. (2020).

[42] Min Xu, Rastislav Bodik, and Mark D Hill. 2003. A "flight data recorder" for enabling full-system multiprocessor deterministic replay. In *Proceedings of the 30th annual international symposium on Computer architecture*. 122–135.

[43] Jingyu Zhou, Meng Xu, Alexander Shraer, Bala Namasivayam, Alex Miller, et al. 2021. Foundationdb: A distributed unbundled transactional key value store. In *SIGMOD 2021*. 2653–2666.

[44] Xinjing Zhou, Xiangyao Yu, Goetz Graefe, and Michael Stonebraker. 2022. Lotus: scalable multi-partition transactions on single-threaded partitioned databases. *Proceedings of the VLDB Endowment* 15, 11 (2022), 2939–2952.