# Rolis: A software approach to efficiently replicating multi-core transactions

Weihai Shen
Stony Brook University

Ansh Khanna
Stony Brook University

Sebastian Angel
University of Pennsylvania
and Microsoft Research

Siddhartha Sen
Microsoft Research

Shuai Mu
Stony Brook University

## Abstract

This paper presents Rolis, a new speedy and fault-tolerant replicated multi-core transactional database system. Rolis's aim is to mask the high cost of replication by ensuring that cores are always doing useful work and not waiting for each other or for other replicas. Rolis achieves this by not mixing the multi-core concurrency control with multi-machine replication, as is traditionally done by systems that use Paxos to replicate the transaction commit protocol. Instead, Rolis takes an "execute-replicate-replay" approach. Rolis first speculatively executes the transaction on the leader machine, and then replicates the per-thread transaction log to the followers using a novel protocol that leverages independent Paxos instances to avoid coordination, while still allowing followers to safely replay. The execution, replication, and replay are carefully designed to be scalable and have nearly zero coordination overhead across cores. Our evaluation shows that Rolis can achieve 1.03M TPS (transactions per second) on the TPC-C workload, using a 3-replica setup where each server has 32 cores. This throughput result is orders of magnitude higher than traditional software approaches we tested (e.g., 2PL), and is comparable to state-of-the-art, fault-tolerant, in-memory storage systems built using kernel bypass and advanced networking hardware, even though Rolis runs on commodity machines.

***CCS Concepts:*** • **Computer systems organization → Reliability**.

***Keywords:*** distributed systems, concurrency, multicore


*EuroSys '22, April 5–8, 2022, RENNES, France*

ACM ISBN 978-1-4503-9162-7/22/04…$15.00
https://doi.org/10.1145/3492321.3519561

**ACM Reference Format:**
Weihai Shen, Ansh Khanna, Sebastian Angel, Siddhartha Sen, and Shuai Mu. 2022. Rolis: A software approach to efficiently replicating multi-core transactions. In *Seventeenth European Conference on Computer Systems (EuroSys '22), April 5–8, 2022, RENNES, France.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3492321.3519561

## 1 Introduction

Transactional storage systems are a key backend component in large-scale online services. They typically come in two flavors: single-machine multi-core databases [20, 21, 24, 38], and distributed and replicated databases [7, 30, 37, 45]. The former tend to be faster and achieve higher throughput owing to the lack of coordination across machines, while the latter achieve important properties such as the ability to continue operating in the presence of crash failures. Recent works [5, 10, 15, 18, 35, 36] attempt to bridge this performance gap by combining multiple machines to achieve fault tolerance, with multiple cores per machine, each of which operates on a different database partition, to achieve high throughput. Despite this progress, we find that they still fall short of the performance achieved by single-machine multi-core databases, owing to the cross-replica coordination needed by distributed transactions.

This paper explores the following research questions: can we significantly improve the throughput of replicated transactions (the bottleneck in making distributed databases perform as fast as multi-core ones) with a more clever coordination protocol between a multi-core leader and its multi-core replicas? And can this protocol finally close the gap between single-machine multi-core and fault-tolerant distributed databases? Our proposed system, *Rolis*, answers both questions affirmatively. Perhaps surprisingly, Rolis's approach to managing and delegating work between a multi-core leader and its multi-core replicas achieves higher throughput than recent works that rely on advanced networking hardware and kernel bypass [5, 10, 18, 36, 41], even though Rolis runs on commodity servers.

Our main idea is to start with a well-known principle: if one maximizes the pipeline of transaction processing and replication by having more transactions outstanding, the

long latency of distributed coordination protocols is roughly masked and the system can achieve similar throughput as a non-replicated transactional system. Of course, just because a principle is well-known does not mean that it can be applied easily. Many designs of distributed transaction systems cannot simply increase the number of outstanding transactions, as this significantly degrades performance. The main reason for this is that the replication protocol is closely tied to the transaction execution and commit protocols. For example, Google's Spanner [7] applies Paxos to replicate the critical steps in two-phase locking and two-phase commit; increasing the number of outstanding transactions will not improve the system's performance, but will instead cause more aborts by increasing the chance of conflicting accesses. More recent protocols like Calvin [37] and Janus [30] partially overcome this limitation, but still fall short of achieving the performance of a single multi-core transactional store.

Rolis's key contribution finds a way out of this conundrum. Rolis uses a combination of speculation and deterministic replay to ensure that we can increase the number of outstanding transactions—maximizing the pipeline and masking the cost of distributed coordination protocols—without causing more aborts. In more detail, Rolis first speculatively executes all transactions on the leader. Then, each thread of the leader creates an independent Paxos stream and uses it to replicate its transaction logs, including the serialization order, to a corresponding thread at each follower. Finally, the followers deterministically replay the logs to arrive at the same state. The main technical difficulties that Rolis overcomes are: (1) how to design a replication protocol that allows each Paxos stream to work independently without coordinating with each other, and without having more outstanding transactions cause aborts; (2) how to replay transactions at the followers to ensure that they can keep pace with the leader; and (3) how to ensure that leader failures do not leave the different Paxos streams in an inconsistent state.

This paper addresses the above challenges and makes the following contributions:

- Identifies the performance gap between distributed transaction systems and multi-core transaction systems that cannot be remedied by merely increasing the number of outstanding requests in traditional approaches.
- Introduces a novel watermark tracking method to eliminate coordination between independent Paxos streams. This watermark is also used for visibility control and safe deterministic replay at followers.
- Describes an implementation and evaluation of Rolis running on a 3-replica setup with 32-core Azure servers that can process 1.03M TPC-C transactions/sec. This throughput is an order of magnitude higher than prior distributed transaction protocols on commodity machines, and it is competitive with recent systems that rely on kernel bypass and RDMA NICs.

The major limitation of Rolis is that it does not (yet) support sharding. While we would like to support a system that scales linearly with more shards, we find that Rolis's throughput, with a single shard, is already equivalent to a system with ~1,000 shards, and should suffice for many applications.

## 2 Overview

### 2.1 Background

**Setup.** In this paper, we study the replication of multi-core transactions. We describe our design and implementation in the context of a key-value storage system, but our design can also apply to other settings, e.g., relational databases and software transactional memory systems. In our setup, a transaction includes multiple read, write, and range query operations that may access different keys. Concurrent transactions that access overlapping keys are isolated, replicated consistently, and obey their real-time order, i.e., transactions are *strictly serializable* [14, 34].

**Consensus and replication.** We assume an *asynchronous* network: messages can be arbitrarily delayed and there is no perfect failure detector that can detect or force a failure in the system. The standard way to achieve consistent replication in an asynchronous network is through consensus protocols such as Paxos [22] or Raft [32]. The standard interface of consensus-based replication is state machine replication (SMR): replicate a sequence of operation logs to all replicas and then have each replica deterministically apply the operations in those logs in the same order.

**Distributed transactions.** The de facto approach for distributing and replicating a transaction across different servers is to use consensus protocols to replicate the critical steps in the transaction execution and commit protocols. Google Spanner, for example, uses two-phase commit (2PC) as a transaction commit protocol and uses Paxos to replicate the critical steps of 2PC, as shown in Figure 1. Other systems may have differences in the protocols they choose, but they all share an important similarity with Spanner: the replication happens side-by-side with the transaction execution and/or commit. That is, replication happens before the transaction's serialization order is determined by the transaction execution/commit protocol.

### 2.2 Problem statement and strawman

In this paper, we wish to answer the question: can we significantly improve the throughput of a replicated transactional key-value store that leverages the many cores available in today's servers, with a more clever coordination protocol between a leader and its replicas?

To answer this question, we start by analyzing a simple strawman that separates replication and transaction execution/commit. In this strawman, a multi-core leader first executes and serializes all transaction requests into a log,

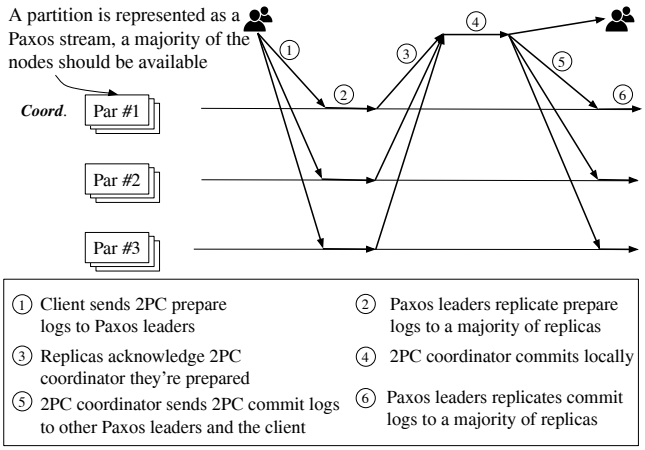



**Figure 1.** Two-phase commit and replication in Spanner [7]. There are three partitions: one serves as a coordinator, the others two as participants.

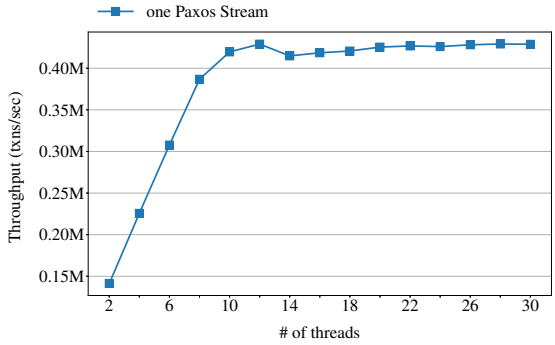



**Figure 2.** Throughput of the TPC-C benchmark in a 3-replica setting with single Paxos stream replication.

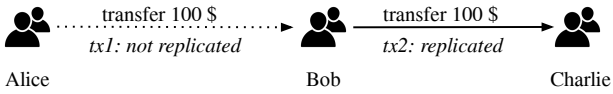



**Figure 3.** A transitive transaction: $tx1$ fails to be replicated and $tx2$ succeeds to be replicated, but $tx2$ depends on $tx1$.

and then uses MultiPaxos to replicate the log to all replicas. When the replication is complete, the leader responds to clients. Each replica then deterministically executes the transactions in the log. Replicas are optimized to use multiple threads to replay the log, which is discussed in more detail in Section 3.4.

Figure 2 shows the performance of this strawman replicated database on the TPC-C benchmark. It is able to achieve over 0.42 million transactions/sec, which is already substantially higher than a single shard in traditional replicate-before-commit approaches. However, throughput plateaus after the leader and replicas use more than ~10 threads. The bottleneck is the single MultiPaxos instance, which we refer to as a *Paxos stream*. All threads on the leader need to write to this Paxos stream, and will thus be bottlenecked by the thread synchronization cost and the replication rate of the Paxos stream. Specifically, when the CPU is fully utilized under a high contention workload with 30 threads, the *enqueue function* (adding outstanding log entries) of the Paxos stream accounts for 68.7% of CPU time.

### 2.3 Challenges and high-level idea

A potential solution to the aforementioned bottleneck is to use more than one Paxos stream. For example, each thread could get its own stream. There are, however, two challenges with using multiple Paxos streams. First, with a single stream we can rely on the natural order of the requests in the stream as the serialization order of transactions, and all replicas can simply follow that order. With more than one stream, requests across streams are not ordered relative to each other, so we need to find another way to ensure that all replicas execute transactions in the same order.

To address this issue, Rolis adopts an execute-replicate-replay model. First, we have one replica (the leader) execute and "commit" the transaction locally as if no replication were to take place, except that the commit is held back from writing real values into the database. Rolis uses the commit operation to determine a serialization order among transactions, which will be captured through a monotonically increasing hardware counter that is available on commodity machines. This timestamp will be encoded into each log and will be used to order requests across different Paxos streams. As we will see, this timestamp can be used to determine dependencies between transactions without the need for coordination among threads.

The second problem is more subtle and difficult to solve. It relates to the failure recovery of the Paxos streams. Suppose that each Paxos stream is independent and therefore recovers independently of the other streams. This could lead to a situation where transactions that are ordered early in the serialization order are lost in recovery, but transactions that are ordered later in the serialization order are preserved. We give an example of this behavior in Figure 3.

Consider two transactions $tx1$ and $tx2$. In $tx1$, Alice transfers $100 to Bob; in $tx2$, Bob transfers $100 to Charlie. $tx1$ happens before $tx2$ in the leader's execution. $tx1$ is replicated by thread-1 in stream-1 and $tx2$ by thread-2 in stream-2. If there is a failure, a possible outcome for the two independent Paxos streams is that stream-1 recovers with $tx1$ not replicated, and stream-2 recovers with $tx2$ replicated. In this case, if replicas only replay $tx2$'s transaction, the system will be in an incorrect state, where Alice never transferred her money out but Charlie received an extra $100.

To address this issue, we need some way to track the dependencies between transactions to make sure that, during

failure recovery, transactions with missing dependencies are not replayed—e.g., in the above example, $tx2$ should not be replayed. This is unfortunate, since our initial goal was to avoid all coordination between threads. Nevertheless, we devise a low overhead mechanism for tracking dependencies that preserves the system's performance. In the context of multi-core transactional systems, low-overhead coordination is well known to be a challenging problem [24, 36]. Rolis's innovation is the use of a lightweight mechanism to track dependencies based on the idea of keeping a *watermark* across all Paxos streams (threads). The watermark is effectively a boundary for replay visibility. All threads will synchronize periodically (with a frequency that does not affect performance) to advance the watermark. All transactions that fall within the watermark are safe to replay, and those beyond the watermark are unsafe to replay. We discuss the details of this and other components of Rolis in the following sections.

## 3 Main System Design

This section describes the main design of Rolis. First, we overview the architecture of Rolis, including its major components and workflow. Then, we describe each stage of the workflow in more detail.

### 3.1 Architecture

Rolis has two major building blocks that we use (almost) as opaque boxes: a multi-core in-memory high-speed transactional database, and a consensus-based replication layer. Our design is generic and is not tied to specific choices for these building blocks. In our implementation, we choose to use Silo [38] as the local database; other options such as ERMIA [20] and Cicada [24] are applicable as well.[1] Silo is a speedy in-memory database that is fast and has good multi-core scalability. It uses an optimized optimistic concurrency control scheme to execute and commit transactions. We adopt Silo's design and interface that the client (application) and database are in the same OS process. For the consensus-based replication, we implemented MultiPaxos following the description given in Chubby [4].

Figure 4 shows the architecture of Rolis. Each replica can run either as a *leader* or as a *follower*. Only a leader can accept new transaction requests. A transaction is first executed on the leader replica, where it is guaranteed isolation from other transactions. The execution will generate a log entry for the transaction, which is assigned a globally unique timestamp that represents the serialization order of the transaction (§3.2). Then, the log entry is replicated to the followers via the replication layer (§3.3). To avoid the scalability bottleneck, the replication layer has multiple Paxos streams, each of which is dedicated to a worker thread in the database.

[1]When using non-strictly serializable databases such as Cicada [24], the isolation guarantee of Rolis downgrades to that of the local database.

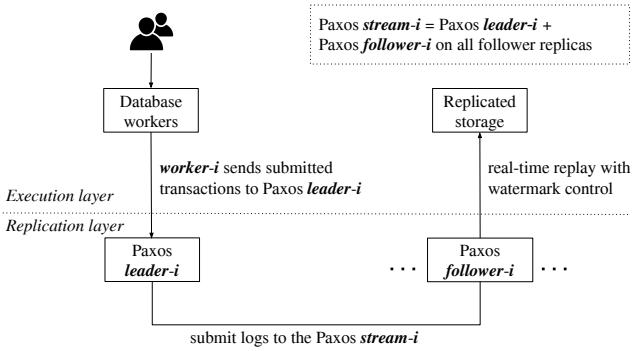



**Figure 4.** Rolis's architecture. There are multiple database workers on the leader replica and each worker has a separate Paxos stream. In this example there are two replicas (one leader plus one follower) and each replica runs a local instance of Rolis.

The Paxos streams are independent from each other in their replication and need not coordinate between them.

When the log entry is replicated to enough followers and is considered durable by the replication layer, the followers will replay the log entry. A follower sees multiple Paxos streams, and replays the log entries in a scalable manner using multiple database threads, ensuring that they are applied in the same order as on the leader (§3.4). We also design a novel scheme (§4.1) to ensure that replay is safe against failures, so that data inconsistencies such as the one described earlier cannot occur. Once the leader confirms that the transaction's log entry *will* be replayed—that is, the leader does not need to wait for the actual replay—the leader can release the transaction's results to the client.

### 3.2 Executing transactions on the leader

Transactions enter the system following the original database's (Silo's) paths, and they are executed up to their commit point in Silo. For completeness, we briefly discuss how Silo works. In Silo, all worker threads are egalitarian and work in a shared memory. A thread starts a transaction and executes it until it finishes. Silo has the standard transaction interfaces consisting of start, read, write, and end. Figure 5 shows an example of these interfaces.

Silo executes transactions using OCC (Optimistic Concurrency Control). During transaction execution, all the reads will record the current versions, and all the writes are buffered in a thread-local workspace. During transaction commit, the worker thread will first (spin-)lock all the keys in the write-set, and then it will validate that the keys in the read-set have not been updated by conflicting transactions, by comparing the most recent versions with the recorded versions. If the validation passes, the transaction will commit and all locks are released. If the validation fails, the transaction will abort and then retry.

```
void *txn = db->new_txn() ;                    // start a transaction
void *table = db->table_instance() ;
int account_alice = table->get("Alice") ;      // read a key
int account_bob = table->get("Bob") ;
assert(account_alice >= 100) ;
table->put("Alice", account_alice - 100);      // write a key
table->put("Bob", account_bob + 100) ;
db->commit_txn(txn) ;                          // end a transaction
```

**Figure 5.** An example of transaction interfaces: Alice transfers $100 to Bob using transaction *start*, *read*, *write* and *end* interfaces.

We make two modifications to Silo. First, if the validation passes, the worker thread will call the timestamp counter instruction, `rdtscp`, to obtain a timestamp, before it unlocks the keys in the write-set. The timestamp obtained from `rdtscp` is monotonically increasing on each core and is synchronized across cores[2]. The performance of the `rdtscp` counter is not a scaling bottleneck for modern OLTP workloads [35], and the overhead is negligible compared to the serialization of transactions. We use this instruction to generate ordered timestamps across threads on the same machine. The timestamp represents a serialization order between conflicting transactions, which the system will utilize to detect dependencies in later replays. In rare cases, two concurrent transactions on different threads may obtain the same timestamp. This is still safe as the two transactions must be non-conflicting, otherwise they would have been blocked from obtaining the timestamp by each other's locks in Silo. After unlocking the write set, the worker thread generates the log for replication, which contains the keys and values of the write set.

For each transaction, Rolis includes each of the modified key-value pairs, but not their individual access times (which Silo uses to keep track of conflicts). Instead, Rolis creates a header for the entire transaction, as shown in Figure 6. This header includes the transaction's timestamp (from `rdtscp`), the epoch number (§3.3), the number of key-value pairs included in the transaction, and the total number of bytes. This information is sufficient to facilitate replay in the follower replicas. To further reduce the overhead of network processing (e.g., interrupts), Rolis batches many transactions (e.g., 1000 transactions in our implementation) into a single log entry. We use the timestamp from the last transaction in the batch to represent the timestamp of the corresponding log entry; this timestamp is compared against the watermark during replay (§3.4).

[2]Synchronization across cores requires CPUs to support `constant_tsc` and `nonstop_tsc` features. Synchronization across sockets requires further support from the motherboard, or can be enforced manually. We use a single socket in our implementation.

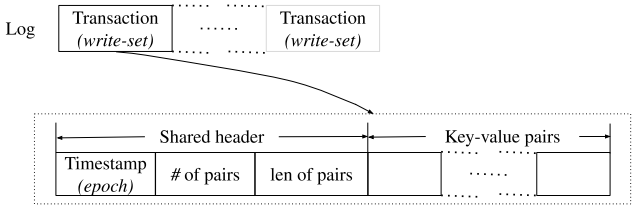



**Figure 6.** Decomposition of batched logs: each log entry contains several transactions and each transaction maintains the shared header and key-value pairs.

The second change we make is holding the worker thread from releasing commit results to the clients until the replication finishes. We use a watermark to control this. Only the results of transactions whose timestamps are smaller than the watermark can be safely released to the clients; transactions whose timestamps are greater than the watermark are still speculative. Section 3.4 explains how the watermark is designed and how it grows so that it is not a bottleneck in the system.

Note that while the worker thread holds off on releasing results to the clients, the thread can still process new requests. In this way, the throughput of the system is preserved and only latency is affected. This introduces another problem in our system design. Since Rolis runs at a high throughput (over a million TPS), there will be (tens of) thousands of transactions whose execution is speculative. In the common case, this is acceptable, because the leader is stable most of the time and the speculative executions will eventually commit. In the worst case, the speculatively executed transactions need to be abandoned. Standard speculative approaches use undo logs to roll back, but in our case, logging undo logs and rolling back thousands of transactions is too costly. Instead, we take a simpler approach. When an old leader rejoins after a crash, it will drop the inconsistent state and join as a new replica (see more discussion in Section 4.3).

### 3.3 Replicating with multiple Paxos streams

Rolis utilizes multiple Paxos streams, based on the intuition that associating each database worker thread with a separate Paxos stream will allow us to maximize scalability while minimizing cross-core synchronization. In general, state machine replication (e.g., Paxos [22] and Raft [32]) is a poor match for multi-core scalability, because SMR usually requires a sequential ordering of all operations. In our implementation, we use the same number of Paxos streams and database worker threads, and then pair each Paxos stream to a database worker. Each Paxos stream only needs to deal with transactions from its associated database worker.

Each Paxos stream receives a log entry from its database worker and replicates the log entry to at least a majority of replicas. After that, the log entry is durable, i.e., the system can recover the log entry from healthy replicas despite a

minority of replica failures. Each Paxos stream reaches agreement independently of the other streams, and thus avoids the cost of coordinating across cores. The leader replica executes transactions speculatively, which are then replicated using each worker's corresponding Paxos stream. Each Paxos stream's log extends progressively with strictly increasing timestamps. Since each Paxos stream on the follower replica has a correct partial order, we can progressively mark consistent snapshots and replay them asynchronously on the followers. In the following section, we describe this replay mechanism in detail.

In order to make it possible to compare timestamps generated by different leaders after a failover, our Paxos implementation uses a monotonically increasing *epoch* number to distinguish a new leader when there is a leader change. This epoch-based method for distinguishing leaders is widely used in consensus algorithms, e.g., it is similar to the term number in Raft. Together, <epoch, timestamp> forms a pair that we can use to serialize all transactions.

### 3.4 Replaying transactions on followers

When a new log entry is durable in the Paxos stream, a follower cannot replay the entry just yet, since this might lead to the inconsistencies discussed in Section 2.3. Instead, we use a watermark scheme to enable safe replay. We avoid the use of explicit dependency tracking [1, 2, 11] for two reasons: (1) explicit dependency tracking (i.e., maintaining a dependency graph) typically involves more expensive and complicated protocols compared to our simple watermark tracking; and (2) explicit dependency tracking may result in cycles, which ultimately result in higher tail latencies [2]. We avoid using a lock-based replay strategy for similar reasons. A lock-based strategy needs to enforce the same lock and unlock ordering recorded on the leader, which essentially requires fine-grained dependency tracking, as in Rex [13].

The watermark tracking works as follows. Recall that we define the timestamp of a log entry to be the timestamp of the last transaction in that log entry. Suppose we have $n$ Paxos streams 1, 2, ..., $n$, for an epoch $e$, and the most recent durable timestamp of log entries in this epoch is $ts_1, ts_2, ..., ts_n$ on a replica. Then the watermark $W_e$ on this replica is:

$$W_e = min(ts_1, ts_2, ..., ts_n)$$

If a log entry in epoch $e$ has its timestamp smaller than $W_e$, all transactions in this log entry are safe from failures. For a leader replica, it is safe to release results corresponding to these transactions to the client. For a follower replica, these transactions are safe to replay. The replay happens as follows. Similarly to transaction execution on the leader, each Paxos stream has a corresponding replay thread. The thread will replay the log entries below the watermark from the stream sequentially. When replaying a transaction, the replay thread will do "compare-and-swaps": it compares the transaction's epoch/timestamp (found in the transaction's header) against those of the keys in the database, to determine if the database should be updated. If a key in the database has a smaller <epoch, timestamp> than the transaction to be replayed, then the thread will update the database to the newer one, including the value, epoch, and timestamp. Otherwise, the thread does nothing and moves on to the next key. In this way, all worker threads can replay their logs concurrently, while ensuring that the latest state is reflected in the database. In our design, we assume that the compare-and-swap is atomic; how this can be done is discussed in Section 5.

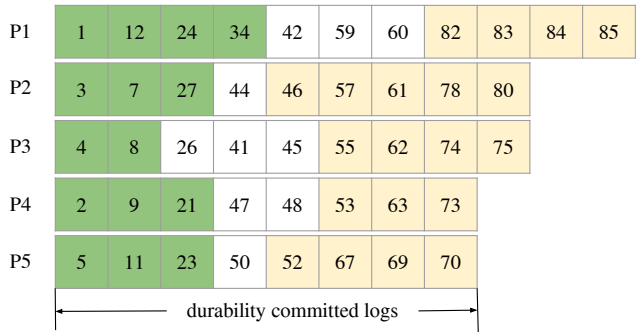



**Figure 7.** Example of logs replicated by five Paxos streams. Logs in the same color are computed within the same interval (0.5ms). Each entry shows the timestamp of the log.

The watermark calculation can be done asynchronously with respect to the log replication. That is, an outdated watermark does not affect safety, because the watermark is always growing and a transaction is always safe to replay once the watermark is beyond the transaction's timestamp. This enables two aspects of our design that make the implementation easier. First, each replica can calculate its watermark independently without any external communication. Second, the system does not need to calculate $W_e$ instantly each time it is accessed, which would incur synchronization across threads. Instead, we calculate $W_e$ periodically, e.g., every 0.5ms as in our evaluation.

**Example.** Figure 7 shows the logs durability committed by five Paxos streams within the same epoch in the first 1.5ms, when $W_e$ is updated at 0.5ms intervals. Rolis advances the watermark from 8, to 44, to 70 at the corresponding intervals. Note that the watermark calculation always happens within the same epoch: each epoch has its own watermark tracking and the system is not allowed to advance the watermark across two different epochs. During an epoch (leader) change, the transactions in the previous epoch may not be safe to replay due to failures. We discuss this further in Section 4.1.

### 3.5 A review of stages in Rolis

One way to look at Rolis is that it separates the "commit points" of transaction processing and consensus processing. We would like to clarify these different commit points,

because the building blocks we rely on have their own definitions of commits, which can be confusing when comparing Rolis to its peers. In fact, in Rolis, there are three separate commit points: in the in-memory database, in the replication layer, and in the overall system.

The first commit point is in the in-memory database (Silo), where the serialization point of the transaction is speculatively determined, which we can refer to as *execution commit*.

The second commit point is in a Paxos stream, where a log is accepted by a majority of replicas and considered durable, which we can refer to as *durability commit*.

The third commit point is when the watermark grows beyond a transaction, so it is safe to release the transaction's results and replay it, which we can refer to as *release commit*.

These commit points occur at the execution stage, replication stage, and replay stage, respectively. Note that the release commit is the real commit point of a transaction; the execution commit and durability commit are not. A transaction can be both execution and durability committed, but aborted by the system in the end.

## 4 Availability

In this section, we discuss how our protocol guarantees fault tolerance without sacrificing either correctness or availability. Rolis needs $2f + 1$ replicas to tolerate $f$ failures. Rolis's Paxos implementation is standard and resembles traditional Paxos systems (e.g., Chubby [3]); we give a brief review here.

Rolis runs an election module on all replicas, which periodically send and receive heartbeats to maintain the leader's liveness in case of machine failures or network partitions. Each replica maintains a single epoch number that indicates the round of election. The follower replicas wait for a time interval, and if no heartbeat is received within this interval, they declare the leader replica as failed and trigger a new election round that increments the epoch number. When a new leader is elected, it first learns all submitted transactions that are durable from the old epoch. Then, a watermark will be computed to determine the boundary between release committed transactions and abandoned transactions (see example in Figure 8). After that, the new leader can receive requests from the application and commit them as normal.

In our Paxos implementation, we use an optimization that prevents a log entry from committing until all previous log entries in the same stream have committed. That is, each stream is always growing sequentially. This simplifies our design because the replay process does not need to deal with any "holes" in the stream where a later log entry commits before an earlier entry.

### 4.1 Replay under failures

Using multiple Paxos streams to support high throughput raises two challenges for correctness. First, as transactions have dependency relationships, Rolis needs to determine

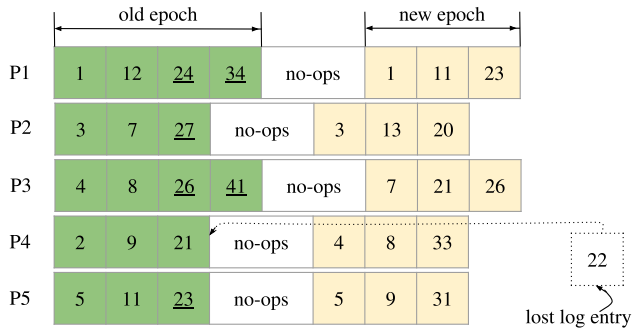



**Figure 8.** An example of failover: durability committed logs in five Paxos streams during a failure. Each entry shows the timestamp of the log.

when and if a transaction can be replayed safely. Second, the Paxos streams on the new leader replica might not have the latest durability-committed transactions locally.

To discuss this in more detail, once a new leader replica has been elected, all Paxos streams on the new leader will start bringing themselves up to date by retrieving any missing durability-committed transactions from other replicas. Then, the new leader commits a no-op in each Paxos stream to end the old epoch, after which it can calculate the latest watermark $W_e$ based on timestamps from the old epoch. Then, Rolis can safely replay all transactions up to $W_e$. Importantly, Rolis can skip transactions above $W_e$ in the old epoch, which is safe for two reasons. First, those transactions have never been executed on other follower replicas due to our watermark scheme. Second, Rolis never releases the results of those transactions to clients on the old leader, or any replica. In effect, the watermark represents the visibility boundary to clients. After replaying transactions in the old epoch, the new leader re-initializes the watermark to continue processing in the new epoch.

**Example.** Figure 8 shows an example of 5 Paxos streams, where all transactions have been durability committed. The timestamps are monotonically increasing in the same epoch within each Paxos stream. Whenever a replay thread sees a no-op, it immediately realizes that membership has changed and then waits for all other replay threads to receive no-ops as well. Once all replay threads observe no-ops, Rolis advances the latest watermark $W_e$ to 21 (the smallest timestamp among: 34, 27, 41, 21, 23) in the old epoch. All replay threads have to replay transactions up to 21, and must skip all remaining transactions (the underlined entries) as they might depend on other transactions that have not durability committed yet. For example, the log entry with timestamp 22 might not have durability committed yet in Paxos stream P4, and will be lost if a failure occurs. If the system replays log entries with timestamps 26 and 41 from Paxos stream P3, the system may become inconsistent because log entries with timestamp 26 and 41 might depend on the lost entry.

Once all replay threads succeed in replaying and skipping transactions in the old epoch, the new leader proceeds as normal in the new epoch (§3.4).

### 4.2 Correctness

We have explained why our mechanisms work. Here, we give a brief review and explain why Rolis guarantees strict serializability across failovers. We separate the discussion into two cases, based on whether the system has replicated all proposed log entries in an epoch before the failover.

We first consider the simpler case when Paxos replication has replicated all log entries in an epoch. Within the same epoch (before a failure occurs), the leader generates a strictly serializable execution whose serial order is captured by the timestamps. This follows directly from Silo's strict serializability guarantees and the way Rolis assigns timestamps. Thus, **transactions with smaller timestamps must be serialized before those with larger timestamps.**

When a failure occurs, the Paxos algorithm guarantees that all replicas see the same Paxos streams, a new leader always has a higher epoch number than the previous leaders, and the epoch numbers divide the stream into log entries generated by different leaders. **Transactions with a smaller epoch number must be serialized before those with a larger epoch number**. This is guaranteed by Rolis's failover, because the new leader does not process requests until it has replayed all transactions from the previous epochs.

Given the <epoch, timestamp> pairs, the replay process will duplicate the final state of the execution, making the replicas consistent with the leader, by always prioritizing the state of transactions with larger epochs/timestamps (via our compare-and-swap method).

We now discuss the more complex case when Paxos replication does not finish replicating all transactions in an epoch, leaving an incomplete tail in the log stream. In this case, the above claims still hold as long as the following premise is true: Rolis can ensure a consistent state by stopping replay once it hits a missing log entry, and ignoring all entries afterwards; i.e., Rolis can replay a prefix of the original execution. In this case, we claim that **Rolis's replay correctly preserves a prefix of the leader's original proposed serializable execution of transactions**. If a transaction is replayed, due to the watermark mechanism discussed in Section 3.4, any transaction that has a smaller timestamp has been or will be replayed. Therefore, the replay always leads to a state that represents a prefix of the original execution. The execution beyond the prefix can be safely discarded because they are never revealed to clients.

Since a new leader finishes the replay of previous epochs before proposing new transactions, the transactions in the new epoch will always read the latest changes.

### 4.3 Adding a new follower replica

In typical SMR implementations, adding a new replica usually requires a snapshot (checkpoint) of the system, which is transmitted to the new replica to bootstrap its state. In our case, Silo does not yet support snapshots: generating snapshots without degrading performance in a multi-core system is challenging, and is not supported by most state-of-art multi-core databases we discussed. So in the spirit of supporting more local database choices (beyond Silo) in our design, we adopt an approach from MongoDB [47] of adding new replicas without using snapshots. Briefly, a new replica first chooses a follower replica as its synchronization source, and performs an asynchronous "pull" from the source by scanning the source's local database, while the source is still working. Then, the new replica will retrieve all logs from the source and replays those logs. After that, the new replica is up to date and can join the replication group. The key reason why this solution works is that the log replay in Rolis is idempotent: repeating the same log entry multiple times does not change the system state.

## 5 Implementation

**Compare-and-swap in replay.** The compare-and-swap operation in the replay process in §3.4 needs to be atomic. In principle, this can be achieved by modifying the underlying data structure of Silo (i.e., Masstree [28]) to support this operation using lock-free instructions or spin-locks. This would achieve better performance but it also requires invasive changes and limits the portability of Rolis to other systems. Instead, we take a slightly slower but more flexible approach: wrapping the compare-and-swap as a Silo transaction. Although this adds overhead to the replay, the replay is still faster than the leader's execution in our tests, as we explain later (§6, Figure 15).

**Heartbeat with empty transactions.** A small issue we have not discussed yet is how the system commits the tail of the log if no new log entries show up. This issue happens when the system is idle or in the process of shutting down. Consider the example in Figure 7, and assume we attempt to stop the system at 1.5ms. The system cannot replay log entries with timestamps above 70 and will wait forever because no new log entries will come in, and the watermark cannot be advanced. To solve this issue, we add a special empty transaction in the heartbeat to every Paxos stream. These empty transactions help the replay finish the transactions hanging at the tail of the log.

**Impact of delayed commit.** In our implementation, we chose a small enough interval (0.5ms) and a reasonable batch size (1000 on the TPC-C benchmark) to advance the watermark, which avoids having too many delayed commit transactions accumulate in memory. In our evaluation of the TPC-C workload, the averaged accumulated memory

is ~0.046 GB with 31 worker threads sustaining a throughput of 1.03M TPS and median latency of 49.41 ms, with an average log size per transaction of 875.6 bytes.

In rare cases, failures such as a network partition can stop the watermark from advancing. In this situation, the system still avoids memory accumulation because the partitioned or failed replicas cannot accept new transactions from Paxos streams, so no transactions will accumulate.

## 6 Evaluation

This section presents our evaluation results, focusing on answering the following questions:

- Can Rolis preserve Silo's performance (including its scalability) in a multi-core setup?
- How does the performance of Rolis compare to that of the state-of-art systems with advanced NIC support, and to traditional software systems?
- How fast can Rolis recover from failures?

### 6.1 Experimental setup

To evaluate the multi-core scalability of Rolis, all experiments were run on multiple Azure virtual machines based on model *Intel Xeon Platinum 8272CL CPU @ 2.60GHz Processor* within the same datacenter. Each machine has 32 (hyper-threaded) CPU cores on a single socket, 128G RAM, and is interconnected via a 16000 Mbps network. Unless otherwise mentioned, the experiments are conducted with 3 replicas. Each trial is run for 30 seconds, the same as Silo's original test configuration. Throughput and latency are calculated based on the release committed transactions. When testing the scalability over threads, we use the `cgroups` kernel feature to limit the CPU and RAM resources. In addition, we always need 1 extra CPU core to advance the watermark and perform leader election tasks, in the event of resource contention with the database workers and Paxos streams.

Like prior work [37, 38, 40], we bind a workload generator within the servers so the "clients" are running in the same address space as Rolis. For completeness and comparison with some baselines, however, some of our experiments add networked clients that issue the requests remotely.

Our experiments run the TPC-C [16] and YCSB++ transactional benchmarks. TPC-C is a common benchmark for OLTP workloads. YCSB++ is a simple transactional benchmark derived from YCSB workload F, which consists of 50% Reads and 50% RMW [6, 43]. The set of records for each transaction in both benchmarks is selected uniformly from the entire database.

Figure 9 shows the percentage and number of read/write operations of each transaction type in TPC-C and YCSB++. For TPC-C, we follow the official ratio for the five transaction types: NewOrder (NEW), Payment (PAY), OrderStatus (ORDER), StockLevel (STOCK), and Delivery (DLVR). We use TPC-C to demonstrate Rolis's performance when handling complicated transactions. For YCSB++, we have two transaction types: Read-Only (READ) and Read-Modify-Write (RMW). The total data space for YCSB++ is 1 million keys (or per partition). We perform 4 updates per RMW operation and 4 reads per READ operation, selecting the items to access at random. We use YCSB++ to push the limit of Rolis with a high throughput transactional workload. The batch sizes we use for TPC-C and YCSB++ are 1,000 and 10,000 respectively.

| TPC-C | NEW | PAY | DLVR | ORDER | STOCK |
|---|---|---|---|---|---|
| Percent | 45% | 43% | 4% | 4% | 4% |
| Get/Scan | avg+23 | avg+3.6 | 50 | avg+3.6 | 3 |
| Insert/Put | avg+24 | avg+4 | 180 | 0 | 0 |

| YCSB++ | READ | RMW |
|---|---|---|
| Percent | 50% | 50% |
| Get/Scan | 4 | 4 |
| Insert/Put | 0 | 4 |

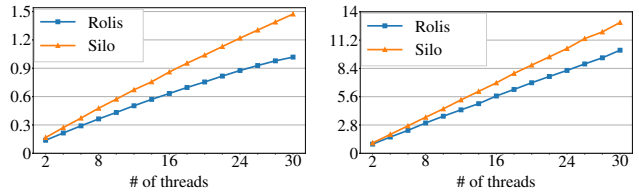
**Figure 9.** Workload of TPC-C and YCSB++ benchmarks. `avg+` stands for an estimated number. Also, we treat scan and get as one read operation, and insert and put as one write operation.

### 6.2 Performance and scalability

We first evaluate the performance and scalability of Rolis over an increasing number of available CPU cores on TPC-C and YCSB++. We also compare Rolis's results to the throughput of Silo on a single machine, i.e., without any replication. Since Rolis is built on top of Silo, the throughput of Silo is the upper bound of our implementation. In Figure 10 and Figure 11, we show that the throughput and per-core throughput of Rolis on both the TPC-C and YCSB++ benchmarks are good and scale well as we introduce more threads.

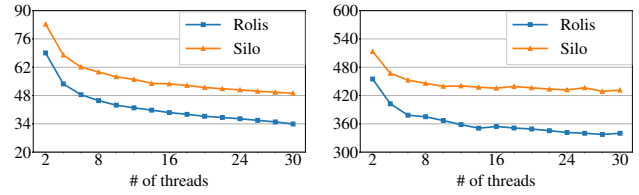
**TPC-C benchmark:** In Figure 10, the throughput of Rolis at 32 cores is 1.03M TPS, which is 68.8% of Silo's throughput. The overhead is mainly caused by transaction (de)serialization and memory copies in the transmission. In our implementation, Rolis needs to serialize/de-serialize 851.8 bytes of data per transaction on average. Figure 11 shows per-core throughput is higher during the first 15 cores, and then it becomes stable gradually, which is the same as Silo. This decreasing tendency is mainly caused by several factors within Silo itself, including increased database size and sharing of resources such as the L3 cache.

**YCSB++ benchmark:** To better study the scalability of Rolis for different workloads, we run an experiment on the YCSB++ benchmark, which is much simpler than TPC-C. In this simple but high throughput scenario, Rolis can still scale well. The throughput using 32 cores can be up to 10.3M TPS, which is 10× compared to the TPC-C experiment. Rolis can retain 77.3% of the throughput of Silo on the YCSB++ benchmark, which is higher than the TPC-C benchmark, mainly because YCSB++ has a smaller write-set.




**(a)** TPC-C **(b)** YCSB++

**Figure 10.** Throughput (million TPS in y-axis) over worker threads on TPC-C and YCSB++ benchmark.




**(a)** TPC-C **(b)** YCSB++

**Figure 11.** Per-core throughput (thousand TPS in y-axis) over worker threads on TPC-C and YCSB++ benchmarks.

### 6.3 Comparison with software implementations

We now turn our attention to how Rolis compares to other transactional key-value stores that are fault-tolerant. In this experiment, we set the number of replicas of each partition to 3. We compare Rolis with 2PL (two-phase locking) using the Paxos-based replication implementation in Janus [30], and also compare with Calvin [37], which is the deterministic concurrency control and replication algorithm implemented in STAR [27], on the YCSB++ benchmark. See Figure 12.

**2PL**. 2PL is the most widely used pessimistic concurrent control protocol. We use the implementation from Janus, a partitioned distributed data store. Each transaction performs 4 read-write accesses or 4 read accesses by incrementing 4 randomly chosen keys. Each partition takes up 1 CPU core for transaction execution. To minimize the extra cost of inter-process communication, we make each transaction only access a single partition. This gives the partitioned test targets an extra advantage over Rolis.

**Calvin**. Calvin uses a central sequencer to determine the order of batched transactions which are sent to all replicas to execute deterministically later. We use the refined Calvin implementation in STAR instead of the original one because (1) the STAR-based implementation has a multi-threaded lock manager instead of a single-threaded lock manager, resulting in better CPU utilization; and (2) STAR's implementation produces more stable and higher throughput. Neither STAR nor the original implementation has replication enabled in their throughput tests, as the design principle of Calvin claims that the replication does not affect throughput. We follow this

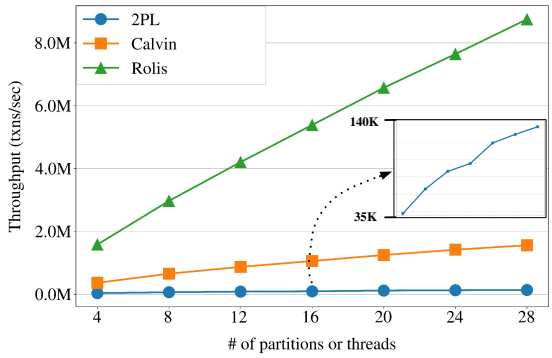



**Figure 12.** Comparisons with traditional software implementations: throughput on YCSB++ benchmark.

in our throughput test. The original Calvin implementation has a ZooKeeper test to evaluate latency under replication. We run this test and report the latency numbers. In term of test setup, we pre-populate each partition with 1 million key-value pairs before starting the experiments. To maximize Calvin's performance, we made two modifications in the setup that give it more advantages. First, all transactions are single-partitioned; there are no transactions that span partitions. Second, we add 4 extra CPU cores for the multiple lock managers and sequencer threads for each experiment run (we are generous and do not count these additional cores during our comparison). In Calvin, transactions are generated in-place on the servers, similarly to Rolis.

In Figure 12, we observe that the throughput of 2PL and Calvin scales correspondingly due to our perfectly partitioned setting. However, 2PL can only achieve a throughput of 137K with 28 partitions. The reasons are twofold: (1) 2PL's implementation is a client-server architecture, and (2) Rolis's OCC-based implementation has more advantages over the 2PL implementation due to the low contention setting. Rolis can achieve a much higher throughput than Calvin and 2PL. Calvin needs a central sequencer to determine the order for a batch of transactions before they start execution, which is expensive compared to Rolis. Similar to Calvin, 2PL also needs intensive coordination among replicas and holds all locks before transaction execution.

### 6.4 Comparison with kernel-bypass systems

We find it hard to compare to hardware-optimized systems in a real evaluation. Many of the state-of-art systems are not open-sourced (e.g., FaRM [10]) or depend on special platform features (e.g., DrTM+R [5] depends on Intel's Restricted Transactional Memory). Meerkat [36] is the only system we found that runs with kernel bypass (DPDK) enabled NICs to compare with. Meerkat targets scaling multi-core performance in replication. It has an advanced fast quorum-based replication and transaction protocol. The replication and

transaction execution are mixed, so it faces the same problem we pointed out that the long latency of replication may compromise performance. Meerkat alleviates this issue by using DPDK to reduce the message latency.

When we try to deploy Meerkat, we find that it is a good example of why a pure software solution like Rolis would have an advantage in maintenance and portability. In our experience, Meerkat does not run out-of-the-box on Azure because the RPC framework Meerkat uses, eRPC, relies on a specific version of the NIC driver, which Azure happens to not support. A newer version of eRPC works on Azure without that driver, but API changes in it make it incompatible with Meerkat. In short, to bring Meerkat into a running state, we spent several extra weeks investigating the issues and made code-level changes to upgrade the eRPC/DPDK library in Meerkat.

For Meerkat tests, we pre-load the entire database with 1 million data items per CPU core to keep the contention level constant as we increase the numbers of cores.

On Azure, DPDK offers a faster user-space packet processing framework by leveraging the advantages of high-performance NICs with FPGA. Our experiments show that the 99.9-th percentile latency between two VMs in the same cluster is ~30 $\mu$s. Figure 13 shows that Meerkat scales to 28 threads and 2.59M TPS on the YCSB-T benchmark, which matches the numbers reported in the Meerkat paper. Meerkat achieves 1.22M TPS with up to 28 cores on the YCSB++ benchmark. In comparison, Rolis achieves up to 7× higher throughput than Meerkat. Note, however, that Rolis uses embedded clients while Meetkat uses networked clients, which gives Rolis an advantage. To check how Rolis performs with networked clients, we implement an (software-based, no DPDK) open-loop networked client to issue stored procedure transactions on the YCSB++ benchmark. The throughput drop after adding networked clients is minor. The main reason is that the RPC library we use is very lightweight and is optimized for batching. For example, with 28 worker threads, only 7% total CPU time is spent on networked servers. Additionally, adding a client saves the cost of request generation on the server (about 2-3% of total CPU time). But note that besides the difference on network stacks, this is still not exactly an apple-to-apple comparison, because Rolis uses stored procedures between the client and server while Meerkat uses interactive transactions.

### 6.5 Failure recovery

We conduct an experiment on replica failures with 3 replicas deployed in the same data center. In each run, we use 4, 8, 16 threads respectively to test failure recovery under different workloads. In the experiment, we kill the leader replica after the system runs for 10 seconds and observe the system's recovery. The recovery time consists of three parts: timeout through heartbeats, leader election, and replaying transactions in the old epoch. Recovery time is largely determined

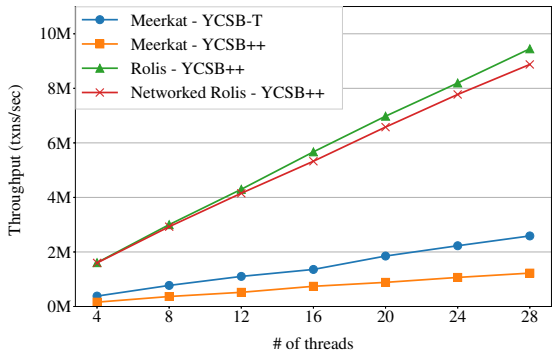



**Figure 13.** Comparisons with Meerkat: throughput on YCSB-T and YCSB++ benchmark.

by the timeout through heartbeats. We set a relatively high timeout, 1 second, to avoid false positive detection and the ensuing expensive recovery process.

After the old leader is killed, a new leader will be elected among live follower replicas and will continue processing requests. As shown in Figure 14, we track the system's throughput in 100 ms intervals and observe a drastic drop at 10 seconds because the leader replica is killed at this moment. Then, the election Paxos component on the follower replica loses heartbeats from the leader and, after a timeout, starts a new round of the election to elect a new leader replica from the live follower replicas. This blocks the system for approximately 1.5–2 seconds; the watermark cannot advance during this downtime.

After a new leader is elected, the throughput of the system quickly climbs to a peak that is higher than before, as the system is trying to commit transactions queued up during the crash. After that, at around 20s, the system returns to a stable level of throughput, which is still slightly better than before the crash happens. This is because the system running on 2 replicas costs less in network communication than running on 3 replicas.

### 6.6 Silo vs replay-only

In this experiment, we conduct an evaluation of Silo versus replay-only, in which we evaluate the throughput of replaying transactions on follower replicas with watermark control and Paxos disabled. This helps us understand the performance of the replay module, and especially whether it would be a bottleneck with the techniques described in Section 5. In this experiment, we pre-generate transaction logs from an independent Silo run, and then load these logs into the replay threads' memory.

Figure 15 shows that the throughput of replay-only at 32 cores is 2.25M TPS, which is 51.5% better than the original Silo's execution. This improvement is mainly because the replay only processes the write-set and ignores the read-set,

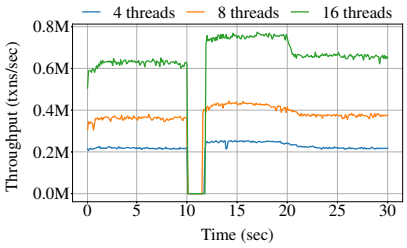



**Figure 14.** Failover test with timeout set to 1s (TPC-C)

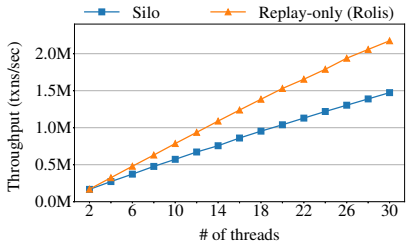



**Figure 15.** The cost of replay-only in the Rolis and Silo (TPC-C)

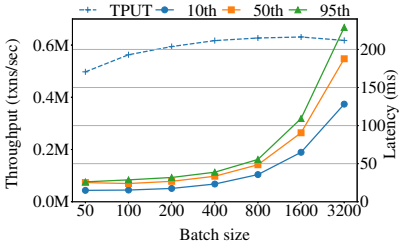



**Figure 16.** Latency and throughput with different batch sizes (16 threads, TPC-C)

which saves much of the cost in the workload. The throughput of replay-only scales very well because there is no complicated synchronization and every key-value update can be performed independently and in parallel. This result indicates that the replay is not the bottleneck in the system.

### 6.7 Skewed workload

This section evaluates the impact of skewed workloads, or "hotspots", on both Silo and Rolis. The skewed workload setting is adopted from the original Silo paper. We run a 100% new-order workload mix and fix the database size to 4 warehouses in a single partition, with the `FastIds` optimization disabled. `FastIds` is an optimization in Silo that generates the id for NewOrder transactions outside of transactions to reduce conflicts, and was enabled in all other tests. We then vary the number of available workers, which simulates increasing workload skew (more workers processing transactions over a fixed size database).

Figure 17 shows that the throughput of Silo stops increasing after 12 workers due to contention on a shared counter per unique (`warehouse-id`, `district-id`) pair, which is used to generate new-order IDs. Rolis can retain 79–82% of the throughput of Silo under this skewed workload, as we expect. The skewed workload is less challenging for Rolis, because Silo has lower throughput in this setup, and less log entries are generated to pressure Rolis's replication and replay.

### 6.8 Batch size versus latency

Batching always induces a trade-off between the throughput and latency of a system: a larger batch size increases the level of parallelism but introduces longer latency. A transaction's latency cannot be lower than the batching duration, which is a major cause for longer latencies. To understand the impact of batching and batch size picking strategy, we conduct an experiment varying the batch size on the TPC-C benchmark using 16 database worker threads.

Figure 16 shows how changing the batch size would affect Rolis's throughput and latency. Compared to a relatively small batch size (50), a batch size of 1,600 can increase Rolis's

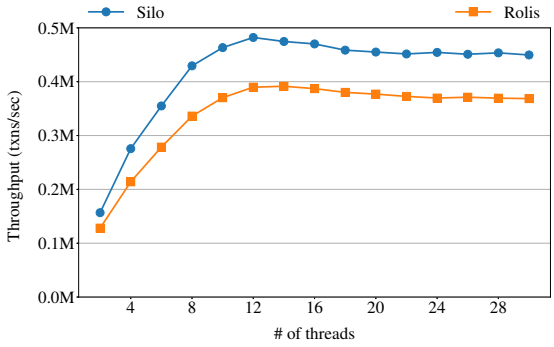



**Figure 17.** Performance of Silo and Rolis under a skewed workload.

throughput by 26.9%. The throughput grows fast as we increase the batch size up to 400, then it keeps increasing at a slower rate until the batch size is 1600. After that, increasing the batch size has an opposite impact on the throughput.

We also display the latency at the 10th, 50th, and 95th percentiles, respectively, as we increase the batch size. The latency of a transaction in this experiment is defined as the duration between the time the client issues a transaction and the time this transaction's results are returned to the client. Figure 16 shows that the latency increases as the batch size grows, as we expect: a higher batch size causes Rolis to take longer to commit a log entry in a Paxos stream, resulting in a longer delay before a transaction goes beyond the advanced watermark to be ready for the replay. Rolis achieves a median latency of 128.2 ms and a 95th percentile latency of 228.9 ms with a high batch size of 3200.

Based on these results, we choose a batch size of 1000 for the TPC-C benchmark and 10,000 for the YCSB++ benchmark in other experiments, balancing good throughput performance with relatively low latency.

We measure the median latency of Rolis, Calvin, and 2PL using 3 replicas within the same data center on the YCSB++ benchmark. Each replica has 16 worker threads. **Calvin**: Latency measured in the original Calvin with ZooKeeper

replication support is 83.01 ms, which comes from three parts: (1) 10-millisecond epoch for batching, (2) sequencers within a replication group use ZooKeeper to agree on a batch of transaction requests for each epoch, and (3) transaction execution. **Rolis**: Similar to Calvin, Rolis's latency is 70.06 ms, which is mainly caused by batching, Paxos streams replication, and asynchronous replay. **2PL**: 2PL has lower latency (21.48 ms) because it does not do batching, resulting in lower throughput.

### 6.9 Factor analysis

To better understand the overheads and benefits of the techniques used in Rolis, Figure 18 shows a factor analysis of performance by breaking Rolis down into cumulative changes. *Silo* refers to the implementation of the Silo protocol as the baseline. *+Serialization* serializes transactions into log entries using `memcpy` operations, as described in Figure 6. *+Replication* replicates log entries to other follower replicas through Paxos streams, as described in Section 3.3. *+Replay* adds replay threads to replay transactions on follower replicas, as described in Section 3.4, and represents Rolis. The workload here is the standard TPC-C mix running with 16 warehouses and 16 worker threads.

**Throughput**. *+Serialization* results in 9.2% throughput loss, which is an unavoidable serialization overhead. *+Replication* causes another 18.1% loss due to two factors: (1) Paxos streams copy serialized log entries into their own log lists, and (2) Paxos streams perform consensus on those logs, which leaves fewer CPU resources for worker threads. *+Replay* has no impact on the throughput because Rolis replays transactions on follower replicas asynchronously.

**CPU**: The leader's CPU is the bottleneck of the system in all cases: the CPU consumption on the leader replica in our tests is always close to 100%. The slight drop in *+Replication* is due to the overhead of adding Paxos threads.

**Memory**: *+Serialization* claims a fixed large memory space per worker thread for the serialized log entries, compared to *Silo*. For *+Replication*, the follower replicas need 12.5GB mainly for the program itself, while the leader replica claims more memory for serialized log entries inside the Paxos streams. *+Replay* does not affect the leader replica but requires another 39.6% CPU for replay and 7.1GB more memory for queued log entries on each follower.

| TPC-C | Silo | +Serialization | +Replication | +Replay (Rolis) |
|---|---|---|---|---|
| Throughput | 863293 | 784103 | 627653 | 631813 |
| cpu - *leader* | 99.1 % | 98.7 % | 96.8 % | 96.9 % |
| mem - *leader* | 17.7 G | 18.7 G | 28.8 G | 28.8 G |
| cpu - *follower* | — | — | 5.1 % | 44.7 % |
| mem - *follower* | — | — | 12.5 G | 19.6 G |

**Figure 18.** Factor analysis for Rolis by adding cumulative components: basic Silo, serialization, replication, and replay.

## 7 Related work

We review related works in the literature on transactional, replicated, multi-core systems from the following aspects.

**Single-machine multi-core transactional systems**. Recent works on optimizing the performance of multi-core transactional systems primarily focus on multi-core scalability in throughput. A large number of them [20, 24, 29, 31, 38, 39, 44] focus on optimizing concurrency control to build multi-core transactional systems. Zen [25] and FOEDUS [21] further discuss optimizations targeting non-volatile memory. All these works are complementary to Rolis as they can potentially replace the local database of Rolis.

While these systems scale well with multiple cores, they cannot be easily extended to a replicated environment, because multi-core execution is inherently non-deterministic, and keeping replicated systems consistent requires each replica to reflect the same order of execution, which is a significant challenge. Rolis uses multiple Paxos streams and the watermark tracking mechanism to address this challenge.

Traditionally, single-machine databases consider improving reliability by supporting checkpoints to disks and recovery from them after rebooting from a crash. This provides a weaker fault-tolerance guarantee than a replication-based solution like ours, because it does not deal with network asynchrony and it usually takes a much longer time to recover. For example, a well-optimized checkpoint approach, SiloR [46], needs several minutes to recover a Silo instance.

**Replicating transactional systems**. There is a line of transactional systems [7, 12, 26, 27, 30, 37, 45, 48] that are optimized to achieve performance and provide availability through replication [42]. Eris [23] and Harmonia [48] both exploit programmable switches by moving concurrency control to switches or detecting read-write conflicts in the network to improve distributed transactions. Some use more sophisticated replication protocols, such as inconsistent replication [45] and asymmetric replication [27] protocols to provide fault tolerance. Deterministic databases [26, 37] are able to efficiently run transactions across different replicas without coordination overhead. Compared to these systems, Rolis achieves a much higher throughput. The major reason is that Rolis is designed to compete against a higher baseline—the multi-core single-machine database—in a replicated setting, and it does so by using an execute-replicate-replay model that masks away the high network latency via intensive pipelining.

A recent popular approach [5, 10, 18, 36] to improve multi-core scalability is using a kernel bypass network. Kernel bypass abstractions (e.g., DPDK, RDMA) are able to provide

both low latency and high throughput. Similar to kernel bypass, NetChain [17] can also significantly reduce the latency of Paxos nodes coordination by caching key-values stores in switches. DrTM+R [5], FaRM [10], and FaSST [18] use kernel bypass to achieve scalability of distributed transactions with primary-backup replication [9, 33] by log shipping. However, they all mix the transaction execution and replication protocols together to commit transactions, since the primary has to receive ACKs from all backups before committing a transaction. Unlike those works, Rolis does not rely on the low-latency provided by advanced hardware to achieve high-throughput. Instead, it decouples the transaction execution and replication protocols carefully to minimize the impact of replication as much as possible.

**Deterministic execution and replay**. Previous works [8, 13, 19, 35] have proposed replicating the scheduling information on the leader to all replicas to make the parallel execution (replay) on followers deterministic. Rex [13] is a multi-core friendly Paxos-based replication system. It allows concurrent execution on the leader replica to proceed freely, while recording the non-deterministic decisions in causally ordered traces. Follower replicas follow the agreed-upon traces by making the same non-deterministic choices in a concurrent replay to reach the same consistent state as the leader. However, as the Rex work pointed out, it is very challenging to replicate a transactional database in this way, because multi-core transactions require too much locking, incurring substantial overhead in tracking the synchronization orders between threads. Eve [19] lets replicas execute commands in parallel speculatively, then replicas verify whether they need to rollback and re-execute the commands in the event of inconsistencies, which incurs high cross-core coordination overhead. Rolis addresses this challenge by choosing not to track the orders between locks, but to track the serialization order at the transaction level at the validation phase of a transaction. This avoids unnecessary coordination overhead in the transaction execution and in the replay.

Scalable-Replay [35] proposed a primary-backup replay-based scheme for replicating a multicore database of Ermia [20]. After executing a transaction, the primary node sends transactions to all backup nodes for replay using a customized multi-version engine on the backups, so that transactions can read and write different versions of rows concurrently. This work has comparable throughput to Rolis. The major advantages of Rolis are two-folded. First, Rolis is a consensus-based approach that deals with network asynchrony and hence provides a stronger fault-tolerance level than a traditional primary-backup approach that assumes synchrony or requires a failure detector. Second, Rolis has a much shorter failover time: Scalable-Replay needs several minutes to do a data conversion on the backups before serving new requests because of different storage structures on the primary and backups, while Rolis only needs 1.5–2 seconds for failover as we show in Section 6.5.

## 8 Conclusion

We present Rolis, a new design for building speedy and fault-tolerant in-memory transactional systems. Rolis adopts an execute-replicate-replay model, and masks the high cost of replication via intensive pipelining. The system is carefully designed so that the execution, replication, and replay are scalable and have nearly zero coordination overhead across cores. Our evaluation shows that Rolis can scale well under different workloads on commodity machines, and achieves high throughput comparable to state-of-art kernel-bypass systems, and orders of magnitude higher than conventional software-only systems.

## Acknowledgments

This paper was substantially improved by detailed comments from our shepherd Martin Maas and the anonymous reviewers of EuroSys '22. We thank Mrityunjay Kumar for leading the development and evaluation of this work in an early stage. We also thank Meerkat authors for providing the code. Mu's group is supported in part by NSF CNS 2130590. Angel's group is supported in part by NSF CNS 2107147, 2124184, 2045861 and DARPA HR0011-17-C0047. The evaluation is supported by Microsoft Azure.

## References

[1] B. Arun, S. Peluso, R. Palmieri, G. Losa, and B. Ravindran. Speeding up consensus by chasing fast decisions. In *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, June 2017.

[2] M. Burke, A. Cheng, and W. Lloyd. Gryff: Unifying consensus and shared registers. In *Proceedings of USENIX Conference on Networked Systems Design and Implementation (NSDI)*, Feb. 2020.

[3] M. Burrows. The chubby lock service for loosely-coupled distributed systems. In *Proceedings of ACM Symposium on Operating Systems Principles (SOSP)*, Nov. 2006.

[4] T. D. Chandra, R. Griesemer, and J. Redstone. Paxos made live: an engineering perspective. In *Proceedings of ACM Symposium on Principles of Distributed Computing (PODC)*, Aug. 2007.

[5] Y. Chen, X. Wei, J. Shi, R. Chen, and H. Chen. Fast and general distributed transactions using RDMA and HTM. In *Proceedings of the Eleventh European Conference on Computer Systems*, Apr. 2016.

[6] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of ACM Symposium on Cloud Computing (SoCC)*, June 2010.

[7] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, et al. Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems (TOCS)*, Aug. 2013.

[8] H. Cui, R. Gu, C. Liu, T. Chen, and J. Yang. Paxos made transparent. In *Proceedings of ACM Symposium on Operating Systems Principles (SOSP)*, 2015.

[9] B. Cully, G. Lefebvre, D. Meyer, M. Feeley, N. Hutchinson, and A. Warfield. Remus: High availability via asynchronous virtual machine replication. In *Proceedings of USENIX Conference on Networked Systems Design and Implementation (NSDI)*, Apr. 2008.

[10] A. Dragojević, D. Narayanan, E. B. Nightingale, M. Renzelmann, A. Shamis, A. Badam, and M. Castro. No compromises: Distributed transactions with consistency, availability, and performance. In *Proceedings of ACM Symposium on Operating Systems Principles (SOSP)*, Oct. 2015.

[11] V. Enes, C. Baquero, T. F. Rezende, A. Gotsman, M. Perrin, and P. Sutra. State-machine replication for planet-scale systems. In *Proceedings of ACM European Conference on Computer Systems (EuroSys)*, Apr. 2020.

[12] H. Fan and W. Golab. Gossip-based visibility control for high-performance geo-distributed transactions. *Proceedings of International Conference on Very Large Data Bases (VLDB)*, 2021.

[13] Z. Guo, C. Hong, M. Yang, D. Zhou, L. Zhou, and L. Zhuang. Rex: replication at the speed of multi-core. In *Proceedings of ACM European Conference on Computer Systems (EuroSys)*, Apr. 2014.

[14] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, July 1990.

[15] C. Hong, D. Zhou, M. Yang, C. Kuo, L. Zhang, and L. Zhou. Kuafu: Closing the parallelism gap in database replication. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. IEEE, 2013.

[16] TPC-C is an On-Line Transaction Processing Benchmark. http://www.tpc.org/tpcc/.

[17] X. Jin, X. Li, H. Zhang, N. Foster, J. Lee, R. Soulé, C. Kim, and I. Stoica. Netchain: Scale-free sub-rtt coordination. In *Proceedings of USENIX Conference on Networked Systems Design and Implementation (NSDI)*, 2018.

[18] A. Kalia, M. Kaminsky, and D. G. Andersen. FaSST: Fast, Scalable and Simple Distributed Transactions with Two-Sided (RDMA) Datagram RPCs. In *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Nov. 2016.

[19] M. Kapritsos, Y. Wang, V. Quema, A. Clement, L. Alvisi, and M. Dahlin. All about eve: Execute-verify replication for multi-core servers. In *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2012.

[20] K. Kim, T. Wang, R. Johnson, and I. Pandis. Ermia: Fast memory-optimized database system for heterogeneous workloads. In *Proceedings of ACM International Conference on Management of Data (SIGMOD)*, June 2016.

[21] H. Kimura. Foedus: Oltp engine for a thousand cores and nvram. In *Proceedings of ACM International Conference on Management of Data (SIGMOD)*, May 2015.

[22] L. Lamport et al. Paxos made simple. *ACM Sigact News*, Nov. 2001.

[23] J. Li, E. Michael, and D. R. Ports. Eris: Coordination-free consistent transactions using in-network concurrency control. In *Proceedings of ACM Symposium on Operating Systems Principles (SOSP)*, Oct. 2017.

[24] H. Lim, M. Kaminsky, and D. G. Andersen. Cicada: Dependably fast multi-core in-memory transactions. In *Proceedings of ACM International Conference on Management of Data (SIGMOD)*, May 2017.

[25] G. Liu, L. Chen, and S. Chen. Zen: a high-throughput log-free OLTP engine for non-volatile main memory. *Proceedings of International Conference on Very Large Data Bases (VLDB)*, Jan. 2021.

[26] Y. Lu, X. Yu, L. Cao, and S. Madden. Aria: a fast and practical deterministic oltp database. *The Proceedings of the VLDB Endowment (PVLDB)*, Aug. 2020.

[27] Y. Lu, X. Yu, and S. Madden. Star: Scaling transactions through asymmetric replication. In *The Proceedings of the VLDB Endowment (PVLDB)*, July 2019.

[28] Y. Mao, E. Kohler, and R. T. Morris. Cache craftiness for fast multicore key-value storage. In *Proceedings of ACM European Conference on Computer Systems (EuroSys)*, Apr. 2012.

[29] S. Mu, S. Angel, and D. Shasha. Deferred runtime pipelining for contentious multicore software transactions. In *Proceedings of ACM European Conference on Computer Systems (EuroSys)*, Mar. 2019.

[30] S. Mu, L. Nelson, W. Lloyd, and J. Li. Consolidating concurrency control and consensus for commits under conflicts. In *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Nov. 2016.

[31] N. Narula, C. Cutler, E. Kohler, and R. Morris. Phase reconciliation for contended in-memory transactions. In *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[32] D. Ongaro and J. Ousterhout. In search of an understandable consensus algorithm. In *Proceedings of USENIX Conference on Annual Technical Conference (ATC)*, June 2014.

[33] D. Ongaro, S. M. Rumble, R. Stutsman, J. Ousterhout, and M. Rosenblum. Fast crash recovery in RAMCloud. In *Proceedings of ACM Symposium on Operating Systems Principles (SOSP)*, Nov. 2011.

[34] C. H. Papadimitriou. The serializability of concurrent database updates. *Journal of the ACM (JACM)*, Oct. 1979.

[35] D. Qin, A. D. Brown, and A. Goel. Scalable replay-based replication for fast databases. *Proceedings of the VLDB Endowment*, Sept. 2017.

[36] A. Szekeres, M. Whittaker, J. Li, N. K. Sharma, A. Krishnamurthy, D. R. Ports, and I. Zhang. Meerkat: multicore-scalable replicated transactions following the zero-coordination principle. In *Proceedings of ACM European Conference on Computer Systems (EuroSys)*, Apr. 2020.

[37] A. Thomson, T. Diamond, S.-C. Weng, K. Ren, P. Shao, and D. J. Abadi. Calvin: fast distributed transactions for partitioned database systems. In *Proceedings of ACM International Conference on Management of Data (SIGMOD)*, May 2012.

[38] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden. Speedy transactions in multicore in-memory databases. In *Proceedings of ACM Symposium on Operating Systems Principles (SOSP)*, Nov. 2013.

[39] Z. Wang, S. Mu, Y. Cui, H. Yi, H. Chen, and J. Li. Scaling multicore databases via constrained parallel execution. In *Proceedings of ACM International Conference on Management of Data (SIGMOD)*, June 2016.

[40] Z. Wang, H. Qian, J. Li, and H. Chen. Using restricted transactional memory to build a scalable in-memory database. In *Proceedings of ACM European Conference on Computer Systems (EuroSys)*, Apr. 2014.

[41] X. Wei, J. Shi, Y. Chen, R. Chen, and H. Chen. Fast in-memory transaction processing using rdma and htm. In *Proceedings of the 25th Symposium on Operating Systems Principles*, Oct. 2015.

[42] M. Wiesmann, F. Pedone, A. Schiper, B. Kemme, and G. Alonso. Understanding replication in databases and distributed systems. In *Proceedings 20th IEEE International Conference on Distributed Computing Systems*. IEEE, 2000.

[43] YCSB open-source implementation. https://github.com/brianfrankcooper/YCSB.

[44] X. Yu, A. Pavlo, D. Sanchez, and S. Devadas. Tictoc: Time traveling optimistic concurrency control. In *Proceedings of ACM International Conference on Management of Data (SIGMOD)*, June 2016.

[45] I. Zhang, N. K. Sharma, A. Szekeres, A. Krishnamurthy, and D. R. Ports. Building consistent transactions with inconsistent replication. *ACM Transactions on Computer Systems (TOCS)*, Dec. 2018.

[46] W. Zheng, S. Tu, E. Kohler, and B. Liskov. Fast databases with fast durability and recovery through multicore parallelism. In *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Oct. 2014.

[47] S. Zhou and S. Mu. Fault-tolerant replication with pull-based consensus in MongoDB. In *Proceedings of USENIX Conference on Networked Systems Design and Implementation (NSDI)*, Apr. 2021.

[48] H. Zhu, Z. Bai, J. Li, E. Michael, D. Ports, I. Stoica, and X. Jin. Harmonia: Near-linear scalability for replicated storage with in-network conflict detection. *Proceedings of International Conference on Very Large Data Bases (VLDB)*, Nov. 2019.

# A Artifact appendix

## A.1 Abstract

A Docker image is provided which contains required dependencies and source code to run the system. In addition, instructions are provided in the `README.md` for 1) running minimal working examples; 2) reproducing the major results.

## A.2 Description & Requirements

**A.2.1 How to access.** The source code is publicly available at https://github.com/stonysystems/rolis. You can either run the binary Docker distribution for validating Rolis's functionalities, or build the system yourself in the real distributed environment.

**A.2.2 DOI.** https://doi.org/10.5281/zenodo.6335844.

**A.2.3 Hardware dependencies.** Rolis is able to run on any x64 server with Docker support. In case you want to reproduce the major results reported in the paper, 3 servers with 32 CPU cores in a single socket are required.

**A.2.4 Software dependencies.** We run all our code on ubuntu 18.04, which mainly depends on common Linux libraries (i.e., boost, gcc and libyaml-cpp-dev). You can install all dependencies by `bash ./install.sh`.

**A.2.5 Benchmarks.** Performance experiments use the TPC-C and YCSB++ benchmark, which are generated by dedicated threads and implemented in `tpcc.cc` and `micro_bench.cc` within the source code, respectively.

## A.3 Set-up

Please follow instructions in the `./README.md` in the source code to set up the system.

**A.3.1 Major Claims.** Rolis can achieve 1.03M TPS on the TPC-C workload, which is comparable to state-of-the-art kernel bypass systems. This is proven by the experiment described in §6.2 whose results are illustrated/reported in Figure 10a.

**A.3.2 Experiments.** You can get all experiment results by running `bash ./one-click.sh`.