COMP 513 - Advanced Computer Systems

Final Project Report - Team ID: 9

Sagar Nandeshwar - Student ID: 260920948

Topic: Rolis: A software approach to efficiently replicating multi-core transactions[1]
Category: Databases and Networking

1. Introduction

Rolis is a fault-tolerant multi-core distributed database system. It aims to maximizing the pipeline of transaction processing and replication, by increasing the number of outstanding transactions, to achieve similar throughput as a non-replicated transactional system, while masking the cost of distributed coordination protocols, i.e. without causing more aborts to increase in in conflicting accesses.

2. Main System Design

2.1 Distributed Environment

In Rolis distributed environment, there will be several servers, of which on will be

Leader Replica:

- Responsible for interreacting with clients and accept transactional request
- Executing transactional request and ensuring the isolation of each execution
- Create Log entry of each execution
- **(These log entries are replicated among the other servers)**

Follower Replica:

- Keep the copies log entries

2.2 Rolis Architecture

Three main components Rolis:

- **Server's Database:** This implemented through Silo. Silo is a multi-core in-memory high-speed transactional database. Here transactions are executed on threads which are dedicated to a worker thread in the database.
- **Replication layer:** Implemented through MultiPaxos. MultiPaxos is the algorithm that run consensus among the servers in unreliable networks
- **Watermark tracking:** To avoid explicit dependency tracking during Paxos while Replaying transactions on followers

3. Baseline Systems

3.1 Silo

(Single Machine) Multi-core in-memory high-speed transactional database.

3.2 2PL (2 Phase Locking)

A client-server architecture that runs Paxos-based replication with two-phase locking.

3.3 Calvin

Central sequencer to determine the order of batched transactions which are sent to all replicas to execute

deterministically later.

4. Set-up

4.1 Hardwar Set-up and Differences

	Paper	Mine	Difference
Cloud Provider	Microsoft Azure	Amazon AWS	-
Processor	3ed gen Intel Xeon Platinum	2nd gen Intel Xeon Platinum	Slightly lower processor
Turbo Frequency	2.60GHz	3.6GHz	138% of the Paper
CPU	32 (hyperthreaded) CPU	32vCPU	same
Memory	128GB	64 GB	50% of the paper
Network	16000 Mbps	8000 Mbps	50% of the paper

Total Cost of ownership = 560.6 CAD

4.2 Software set up

For creating a distributed environment there were three AWS EC2 c5a.8xlarge deployed. Each have

- Docker Container
- Rolis Package
- Knowledge of other EC2 instance

In order to set-up the above requirement I had to,

- Set up Docker Container
- Set up Rolis Packages
- Establish Connection between each server (such ssh IP works)
- Change EC2 Configuration

→ Remove root logins
→ Remove password authorization
→ allow port 22 traffic
→ allow empty password
→ network access without (.pem)
→ set up password and username
→ gave public IPs

- Change DockerFile

→ get EC2 permission keys
→ Change host names
→ change network permissions

- Network security and interference group

→ Allow in bound traffic without authorization
→ Make subnet of the three instances. (Does not affect the network speed)

- Codes Modification

→ scp, to manually moves modified file to other servers to synchronize network setting
→ change file permissions

4.3 Testing

Rolis has **one-click.sh** to run experiments for Rolis, Calvin and 2PL.

- It produces a Text Log under results folder.
- It takes 4 hours for Rolis experiments, 0.5 for Calvin experiments and 1 hour for 2PL experiments.

5. Experiments

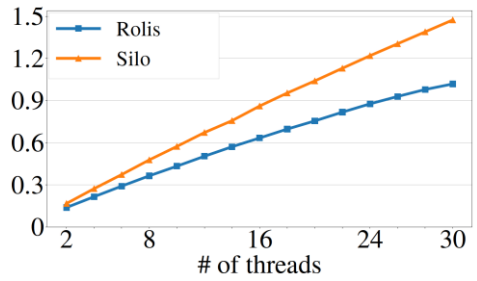
5.1 Performance and Scalability

Evaluating Rolis performance on TPC-C and YCSB++ benchmarks and comparing it against Silo's performance.

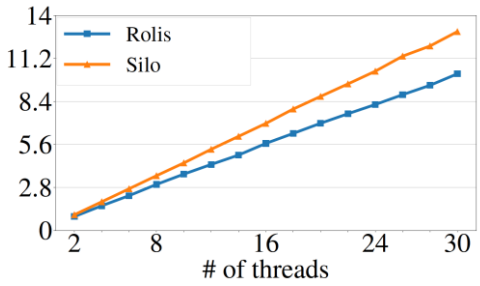
Motivation:

- To Evaluate Rolis performance when handling complicated transactions (TPC-C).
- To test the limit of Rolis on heavy workload (YCSB++)
- To compare Rolis performance against Single Machine (non-replicated) Database (Silo)

Reproducing: Section 6.2 Performance and scalability Figure 10. Throughput (million TPS in y-axis) over worker threads on TPC-C and YCSB++ benchmark.



Paper: Throughput (million TPS in y-axis) over worker threads on TPC-C benchmark.

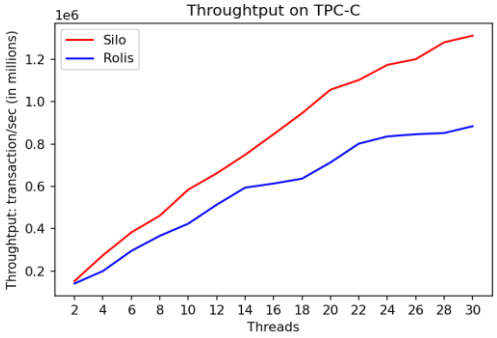


Paper: Throughput (million TPS in y-axis) over worker threads on YCSB++ benchmark.

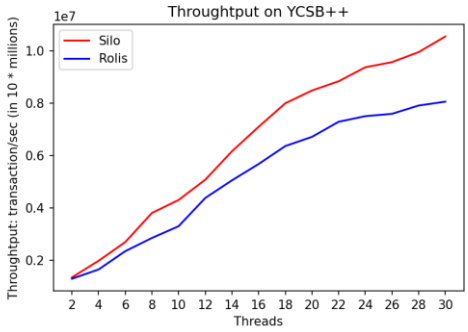
Two benchmarks are as follows,

- **TPC-C Dataset:** Consist of complex transactions, such as NewOrder (NEW), Payment (PAY), OrderStatus (ORDER), StockLevel (STOCK), and Delivery (DLVR).
- **YCSB++ Dataset: Consist** of simple Read-Only (READ) and Read-Modify-Write (RMW). There are 50% Reads and 50% RMW request, and in total 1 million keys and pairs.

Result:



Throughput (million TPS in y-axis) over worker threads on TPC-C benchmark.



Throughput (million TPS in y-axis) over worker threads on YCSB++ benchmark.

The gap between blue (Rolis Performance) and red (Silo Performance) line (majorly) represents latency due to replication layer.

In our experiments we see a very similar trend to the paper in the performance of Rolis for both TPC-C and

YCSB++ benchmarks. Rolis in the paper achieved 68.8% and 77.3% of Silo's throughput on TPC-C and YCSB++, whereas we achieved 59.3% and 70.1% on PC-C and YCSB++ of Silo's throughput, respectively. This is because I had comparable hardware set-up as in the paper.

For the TPC-C and YCSB++ benchmarks, we notice that the performance of Rolis and Silo's dipped a little as we increase the number of threads, compared to the paper. Since transactions are executed on threads, we suspect that the dip in the performance is due to lower processing power and half memory size.

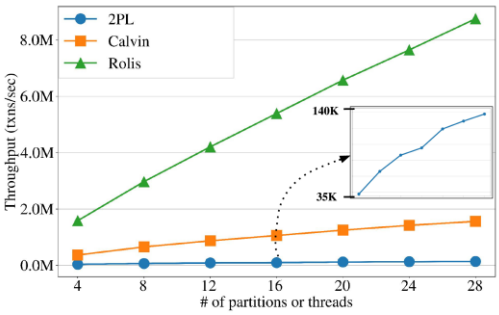
We also notice that for both TCP and YCSB++, the throughput increase is not as linear as presented in the paper; it fluctuates throughout the graph. This may be due to much slower network speed (nearly 50%). Also, modifications in network configuration, such as passing keys (.pem) and relaxing the security group setting of EC2 may have also caused more contention, further un-stabilizing the network speed.

5.2 Comparison with Software Implementations

Comparing Rolis performance against 2PL and Calvin.

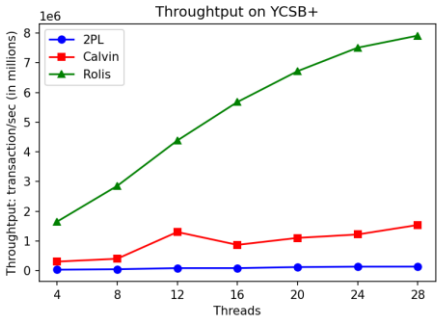
Motivation: Compare Rolis performance against state-of-art transactional databases

Reproducing: Section 6.3 Comparison with software implementations. Figure 12. Comparisons with traditional software implementations: throughput on YCSB++ benchmark.)



Paper: Comparisons with traditional software implementations: throughput on YCSB++ benchmark.

Result:



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

We again see similar trends in the throughput of Rolis, as well as Calvin and 2PL, as in the paper. Rolis has outperformed 2PL and Calvin. This was expected as 2PL follows server-client architecture, whereas Rolis OCC-based implementation, this causes Calvin to have more contentions as compared to Rolis. On the other hand, Calvin needs a central sequencer to determine the order for a batch of transactions before they start execution which is expensive compared to Rolis.

Here we again observe that the throughput of Rolis dipped as we increased the number of threads as compared to

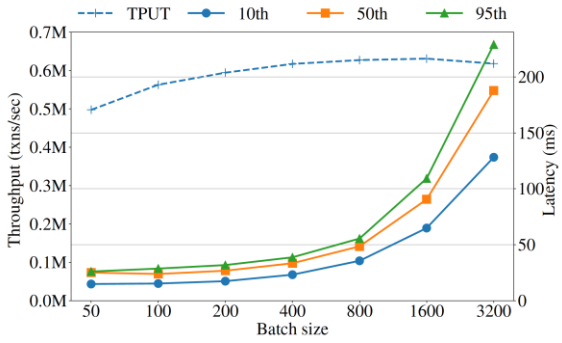
the paper. We also notice that none of the systems were able to reach as high throughput as given paper for large no. of threads. We again suspect that this is due to limited computation power, lower network speed and lower memory.

5.3 Batch size versus Latency

Analyzing the effect of log batching on throughput and latency.

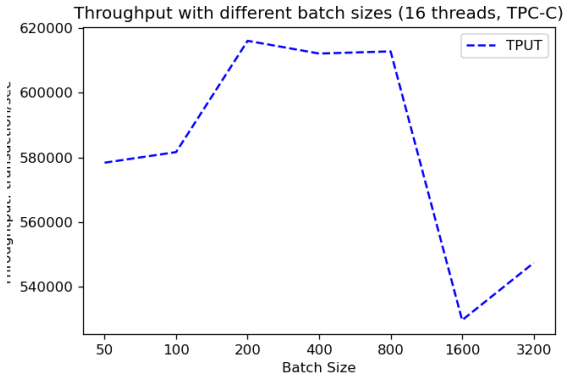
Motivation: Analyzing throughput vs latency trade-off due to log batching.

Reproducing: Section 6.8 Batch size versus latency Figure 16. Latency and throughput with different batch sizes (16 threads, TPC-C))

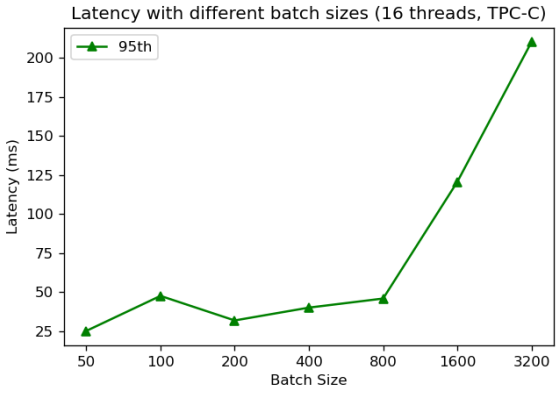


Paper: Latency and Throughput with different batch sizes (16 threads, TPC-C)

Result



Throughput with different batch sizes (16 threads, TPC-C)



Latency with different batch sizes (16 threads, TPC-C)

Except batch_size if 100 and 1600, the throughput we receive is very similar the paper. The throughput, as expected, increases as we increase the size of log batch from 50 to 800, and then decrease after 800. However, for batch_size 100 and 1600 we regularly received an irregular throughput. For batch_size 100, I did not see any significant difference during the experiment, but for batch_size 1600, I do observe a significantly longer wait on leader replica, than other batch_size. We can also see that the latency for batch_size 100 and 1600 is a bit higher than expected.

We know that as we increase the batch_size the performance increases due to parallelization, however this also introduces longer latency. This may become more significant at batch_size 1600, due to lower RAM and processing power. For batch_size 100 we say that the batch_size is too small for 16 threads running TCP commands as they are comprising of complex request, demanding more processing time, because of which transaction have longer latency before execution.

6. Non-Reproduced Experiments

Of the all the 8 experiments, I was able to run 7 of them. Rolis runs inside the docker container, therefore I did not face any software issue. Moreover, I had comparable hardware set up and network interface. Because of this Rolis did not timeout or crash during any experiments.

In terms of the performance, I was able to reproduce similar results for Silo vs replay-only and Skewed workload. This may be due to fact that in Silo vs replay-only, we evaluate the throughput of replaying transactions on follower replicas with watermark control and Paxos disabled, therefore there is no complicated synchronization, and every key-value update can be performed independently and in parallel. Since this experiment mostly depends on processing power and parallelization, and I had comparable hardware, I was able to achieve similar results. In Skewed workload, we evaluate the performance of Rolis and Silo in skewed workloads, which also mostly depend on the processor. Also, since silo has lower throughput in this setup, and less log entries are generated, the network component does have much impact on the throughput.

For the failure recovery, the experiment hugely depends on the network set-up. From experiment 1 and 2 we can observe some significant fluctuation in throughput, which may be due to slower network and EC2 contentions. I observe that it takes much longer to recover from the failure as compared to the paper. This may be due to the fact that this experiment depends on the latency due timeout through heartbeats, leader election, and replaying transactions in the old epoch which majorly depends on the network speed.

I did not complete the Comparison with kernel-bypass systems experiment. The kernel-bypass requires the implementation of hardware-optimized systems and many of the state-of-art systems are not open-sourced. It was also difficult to implement Meerkat on AWS instance. I also believe that since the replication and transaction executions are mixed, it would long latency of replication may compromise performance, which would produce different results than the paper.

Reference:

[1] “Rolis: A software approach to efficiently replicating multi-core transactions” - Weihai Shen, Ansh Khanna and Sebastian Angel