SILO: SPEEDY TRANSACTIONS IN MULTICORE IN-MEMORY DATABASES

1

Stephen Tu, Wenting Zheng, Eddie Kohler[†], Barbara Liskov, Samuel Madden

Presenter : Akshada Kulkarni

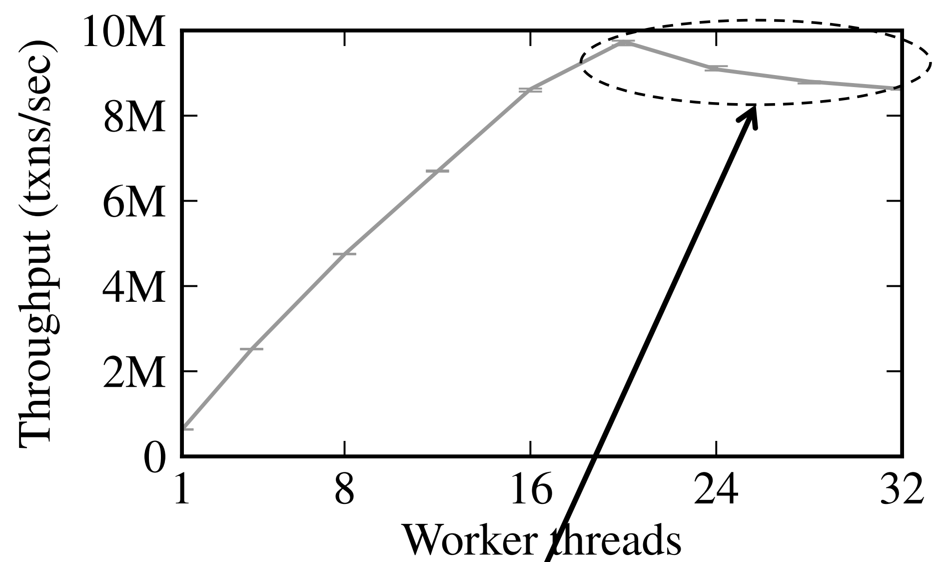
Acknowledgement : Author's slides are used with some additions/ modifications

TABLE OF CONTENT

- Introduction
- Design
- Evaluation
- Conclusion

INTRODUCTION

MULTICORES TO THE RESCUE?




```
txn_commit()
{
    // prepare commit
    // […]
    commit_tid = atomic_fetch_and_add(&global_tid);
    // quickly serialize transactions a la Hekaton
}
```

SILO: TRANSACTIONS FOR MULTICORES

- Near linear scalability on popular database benchmarks.
- uses minimal-contention serializable and scalable commit protocol.
- achieves roughly 700,000 transactions(OLTP) per second on the standard TPCC benchmark on a single 32-core machine, i.e. about 22,000 transactions per second per core[4].

SECRET SAUCE

- A scalable and serializable transaction commit protocol.
  - Shared memory contention *only* occurs when transactions conflict.
- OCC maintains local read and write sets and writes only at commit time after validation.
- Scalability achieved by eliminating unnecessary contentions
- Recovery is possible using a form of epoch based group commit

DESIGN

EPOCHS

- Divide time into epochs.
  - A single thread advances the current epoch.
- Use epoch numbers as recovery boundaries.
- *Reduces* non data driven shared writes to happening very infrequently.
- Serialization point is now a memory read of the epoch number!

TRANSACTION IDENTIFIERS (TIDS)

- Each record contains TID of its last writer.
- TID is broken into three pieces:

Status bits	Sequence number	Epoch number

0 63

- Assign TID at commit time (after reads).
  - Take the smallest number in the *current* epoch larger than all record TIDs observed in the transaction.

PRE-COMMIT EXECUTION

- **Idea**: proceed as if records will not be modified – check otherwise at commit time.
- Maintain *read set* : records that are read with TIDs
- Maintain *write set* : new state of the record (not the previous TID)
- (Standard optimistic concurrency control)

COMMIT PROTOCOL

- **Phase 1**: Lock all records in the write set by acquiring the record's lock bit.
  - Read the current epoch. (Fences are required)
- **Phase 2**: Validate records in read set.
  - Abort if record's TID changed or lock is held (by another transaction).
- **Phase 3**: Pick TID and perform writes.
  - Use the epoch recorded in Phase 1 for the TID.

RETURNING RESULTS

- Say T1 commits with a TID in epoch E.
- Cannot return T1 to client until all transactions in epochs $\leq$ E are on disk.

CORRECTNESS

- Locks all written records before validating TIDs of read records
- Treats locked records as dirty and aborts on encountering them
- Fences ensure that TID validation checks all concurrent updates
- Epoch number is serialization point.
- One property we require is that epoch differences agree with dependencies.
  - T2 reads T1's write → T2's epoch ≥ T1's.
  - T2 overwrites a key T1 read → T2's epoch ≥ T1's.

WRITE-AFTER-READ EXAMPLE

- Say T2 overwrites a key T1 reads.

Time

```
T1:
tmp = Read(A);
WriteLocal(B, tmp);

Lock(B);
e = Global_Epoch;

Validate(A); // passes
t = GenerateTID(e);

WriteAndUnlock(B, t);
```

```
T1() {
    tmp = Read(A);
    Write(B, tmp);
}
```

```
T2() {
    Write(A, 2);
}
```

T2's epoch ≥ T1's epoch

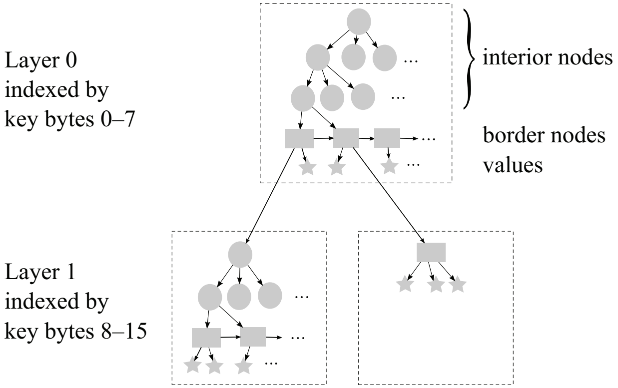
```
T2:
WriteLocal(A, 2);

Lock(A);
e = Global_Epoch;

t = GenerateTID(e);

WriteAndUnlock(A, t);
```

B - - - → **A**

A *happens-before* B

STORING THE DATA

- A commit protocol requires a data structure to provide access to records.
- We use Masstree, a fast *non-transactional* B-tree for multicores.



EVALUATION

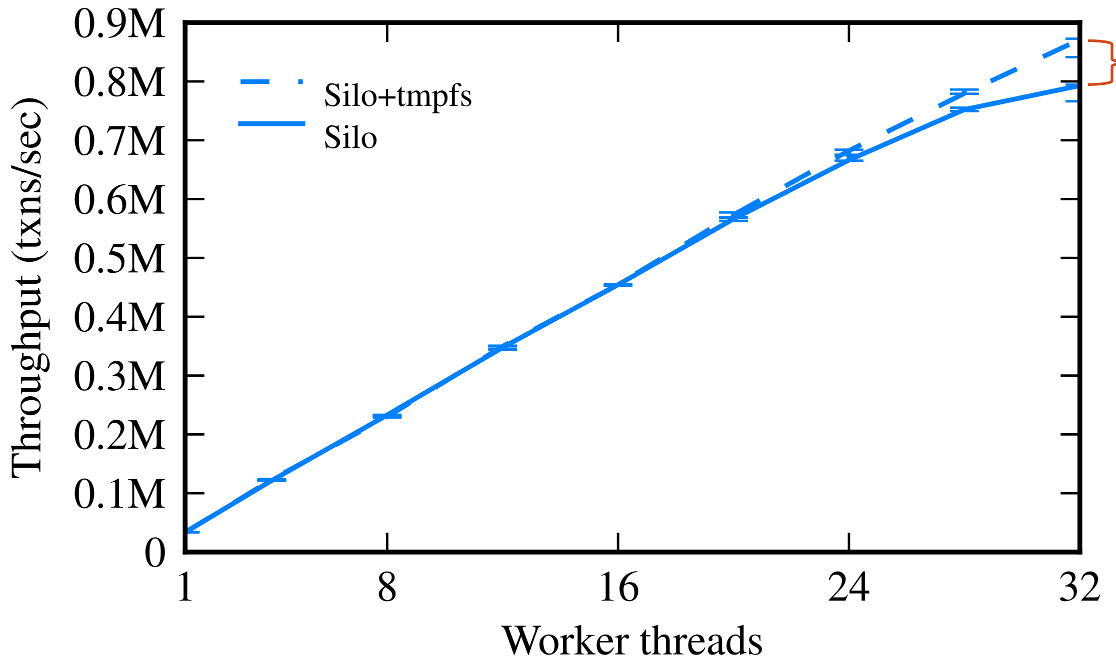
SETUP

- 32 core machine:
  - 2.1 GHz, L1 32KB, L2 256KB, L3 shared 24MB
  - 256GB RAM
  - Three Fusion IO ioDrive2 drives, six 7200RPM disks in RAID-5
  - Linux 3.2.0
- No networked clients.

WORKLOADS

- **TPC-C**: online retail store benchmark.
  - **Large transactions** (e.g. delivery is ~100 reads + ~100 writes).
  - Average log record length is ~1KB.
  - All loggers combined writing ~1GB/sec .
- **YCSB-like**: key/value workload.
  - **Small transactions.**
  - 80/20 read/read-modify-write.
  - 100 byte records.
  - Uniform key distribution.

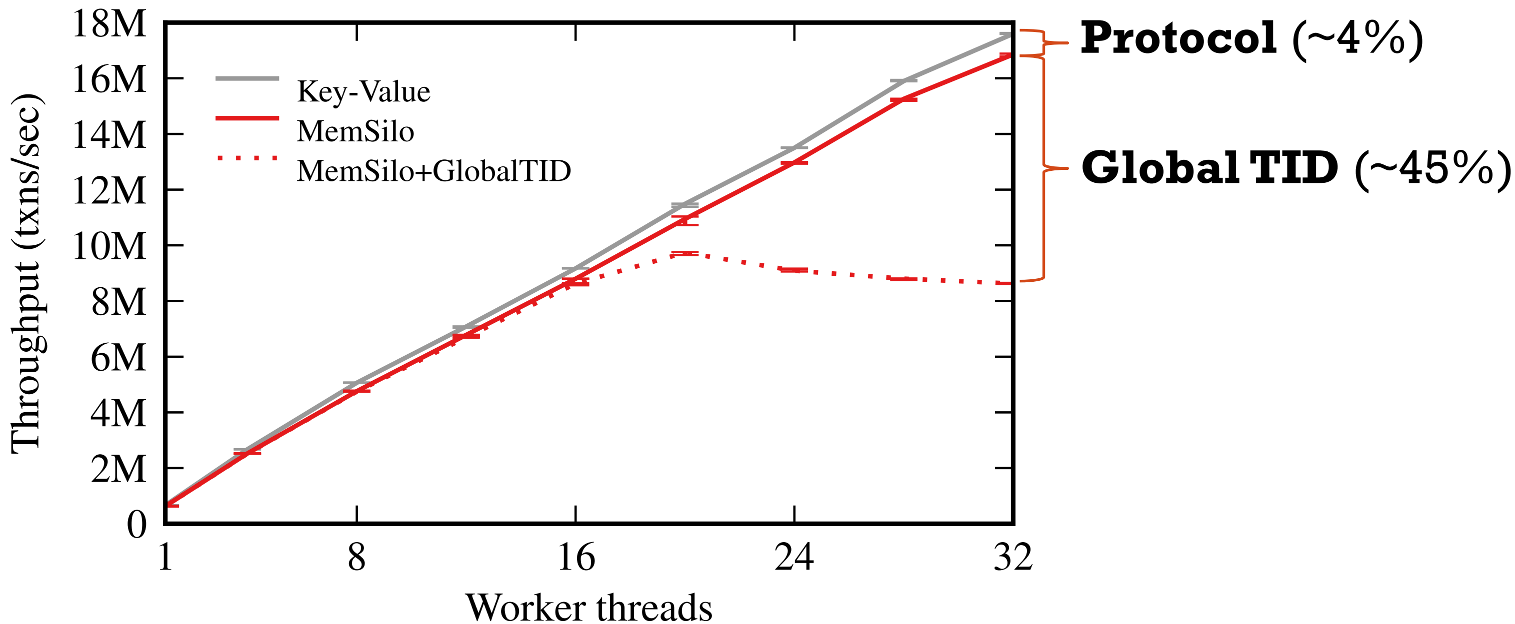
SCALABILITY OF SILO ON TPC-C



I/O
(scalability bottleneck)

- I/O slightly limits scalability, protocol does not.
- Note: Numbers several times faster than a leading commercial system + numbers better than those in paper.

COST OF TRANSACTIONS ON YCSB



- Key-Value: Masstree (no multi-key transactions).
  - Transactional commits are inexpensive.
- MemSilo+GlobalTID: A single compare-and-swap added to commit.