WISCONSIN
UNIVERSITY OF WISCONSIN–MADISON

CS 839: Design the Next-Generation Database

Lecture 5: Multicore (Part II)

Xiangyao Yu

2/4/2020

Announcements

Will upload slides before lecture

Submit review to see others' reviews

Computation resources:

- CloudLab: https://www.cloudlab.us/signup.php?pid=NextGenDB
- Chameleon: https://www.cloudlab.us/signup.php?pid=NextGenDB
- AWS: Apply for free credits
  at https://aws.amazon.com/education/awseducate/

Discussion Highlights

T/O vs. 2PL

- **Pros of T/O**: simpler, no deadlocks
- **Cons of T/O**: timestamp allocation bottleneck, storing read/write timestamps
- **Examples of timestamps**: third-party authentication, sync/recovery in distributed systems, consensus, parallel compilation, cryptocurrency, network protocol, cache coherence, distributed file systems, transactional memory, vector-clock

Multi-versioning

- **Pros**: good for rolling back, efficient writes, non-block reads
- **Cons**: Memory copy, memory space, timestamp bottleneck
- **HTAP**: old versions for OLAP and new versions for OLTP

Hardware for concurrency control

- NUMA management, clock synchronization, low-overhead locking, timestamp allocation, conflict detection in hardware, Persistent memory for logging, locking prefetch

Hyper uses fork() for consistent snapshot

- HyPer: A Hybrid OLTP&OLAP Main Memory Database System Based on Virtual Memory Snapshots, ICDE 2011

Today's Paper

Speedy Transactions in Multicore In-Memory Databases

Stephen Tu, Wenting Zheng, Eddie Kohler[†], Barbara Liskov, and Samuel Madden
MIT CSAIL and [†]Harvard University

Abstract

Silo is a new in-memory database that achieves excellent performance and scalability on modern multicore machines. Silo was designed from the ground up to use system memory and caches efficiently. For instance, it avoids all centralized contention points, including that of centralized transaction ID assignment. Silo's key contribution is a commit protocol based on optimistic concurrency control that provides serializability while avoiding all shared-memory writes for records that were only read. Though this might seem to complicate the enforcement of a serial order, correct logging and recovery is provided by linking periodically-updated epochs with the commit protocol. Silo provides the same guarantees as any serializable database without unnecessary scalability bottlenecks or much additional latency. Silo achieves almost 700,000 transactions per second on a standard TPC-C workload mix on a 32-core machine, as well as near-linear scalability. Considered per core, this is several times higher than previously reported results.

nization scale with the data, allowing larger databases to support more concurrency.

Silo uses a Masstree-inspired tree structure for its underlying indexes. Masstree [23] is a fast concurrent B-tree-like structure optimized for multicore performance. But Masstree only supports non-serializable, single-key transactions, whereas any real database must support transactions that affect multiple keys and occur in some

SOSP 2013

A Simple Optimistic Concurrency Control

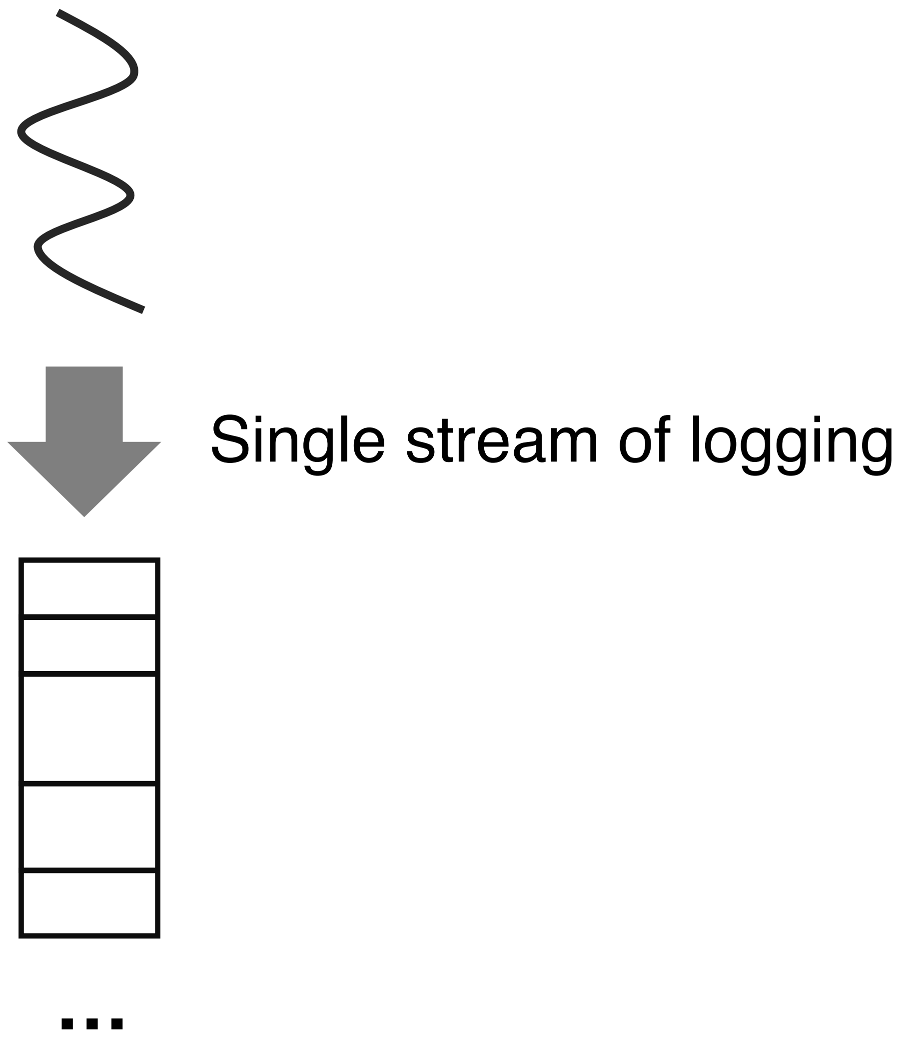
```
// read phase
read(): read record into read set (RS)
update(): read record into write set (WS) and update local copy

// validation phase
validation():
    lock all records in WS
    for r in RS ∪ WS:
        if r.version != DB[r.key].version or r.is_locked:
            abort()

// write phase
write():
    for r in WS:
        DB[r.key].value = r.value
        DB[r.key].version ++
        unlock(r)
```

Properties of This OCC

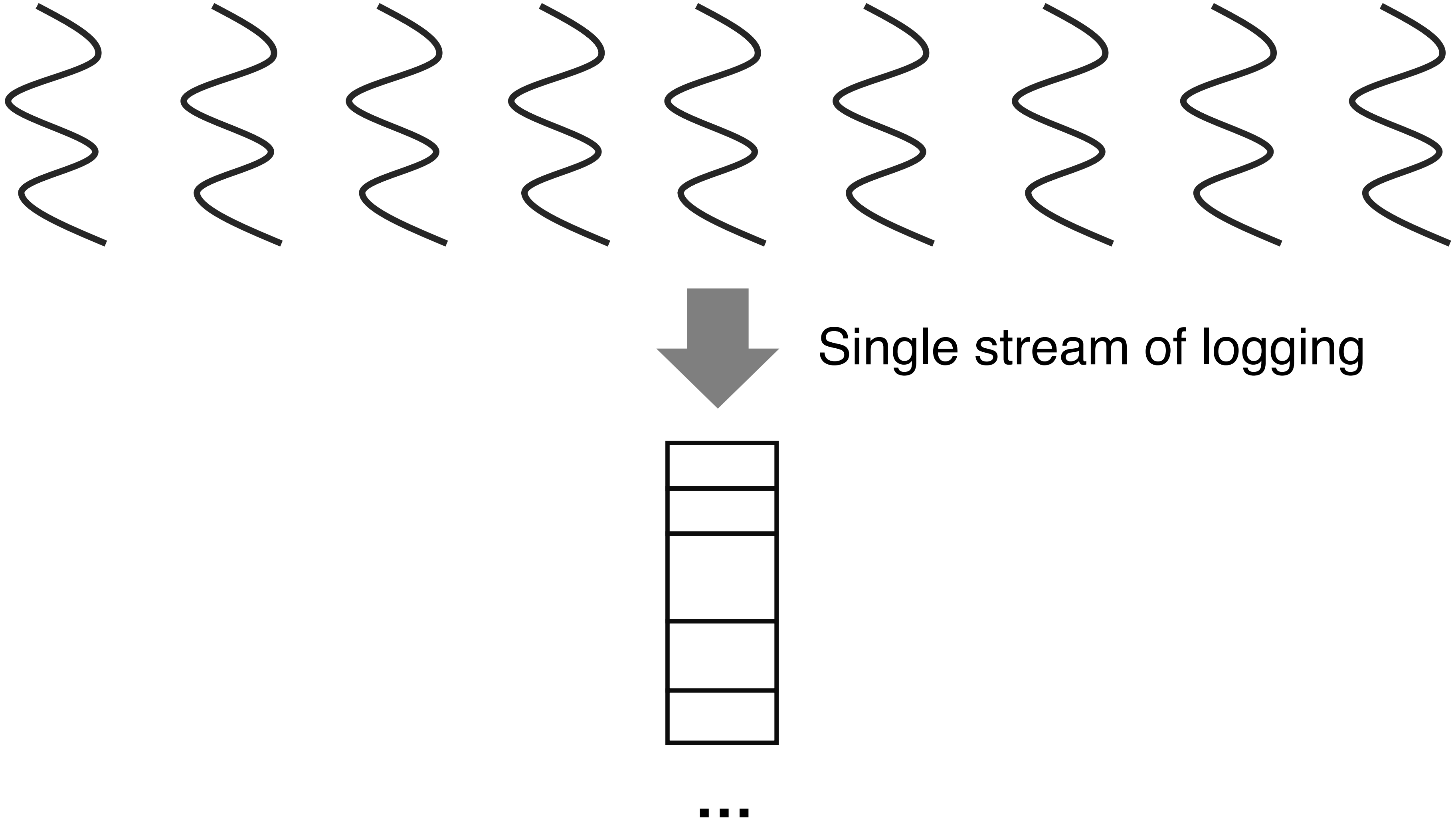
High Scalability: No scalability bottleneck for workloads with no contention

Invisible Read: reads do not write to shared memory

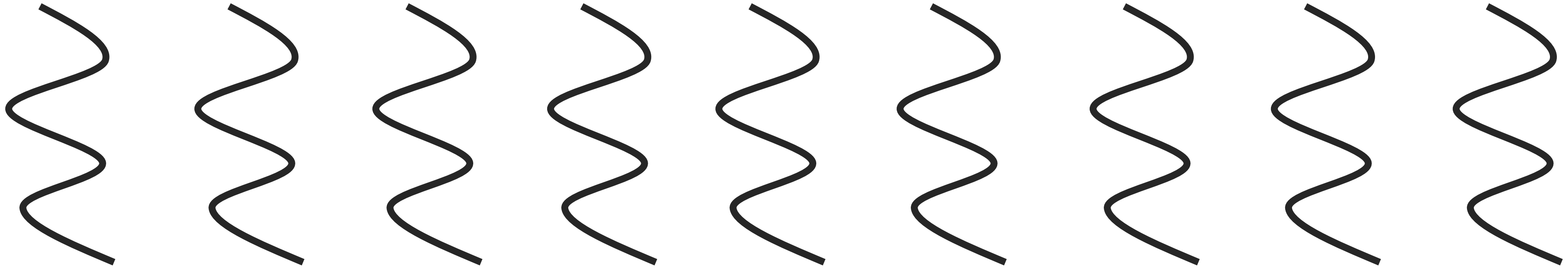
Scalability of Logging

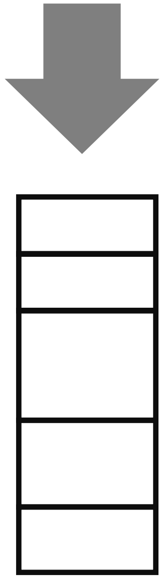


Scalability of Logging



Scalability of Logging





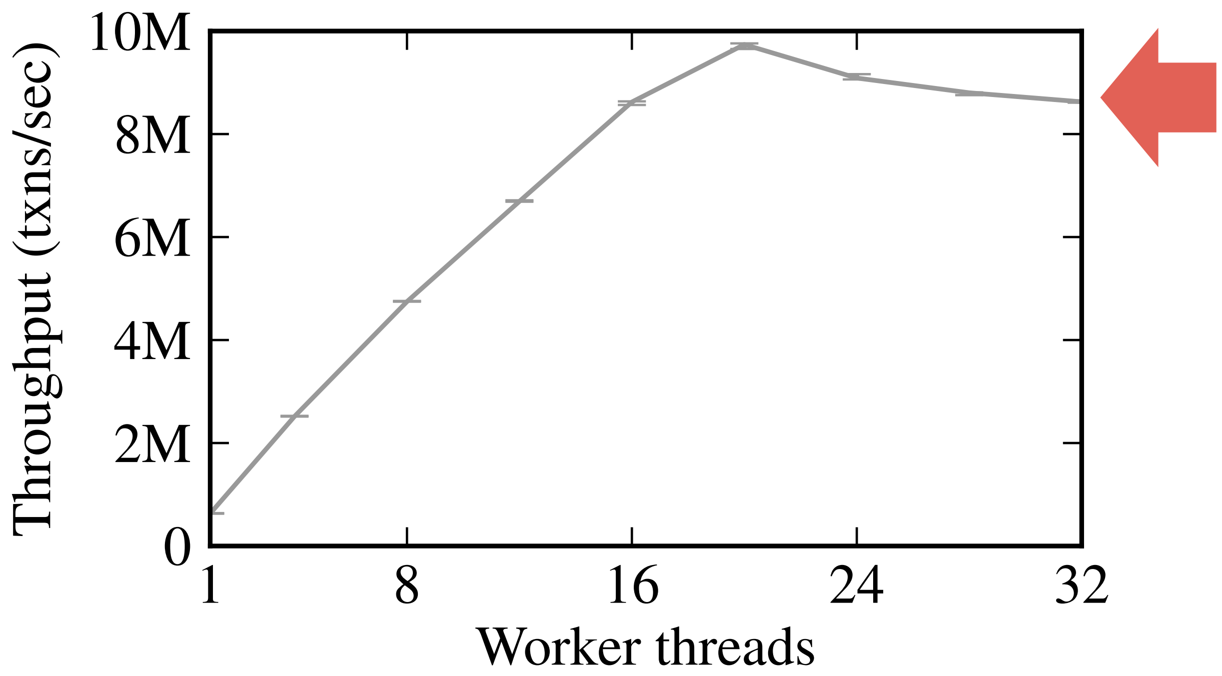
Single stream of logging

```
Log_Sequence_Number =
atomic_fetch_and_add(&lsn, size);
```

...

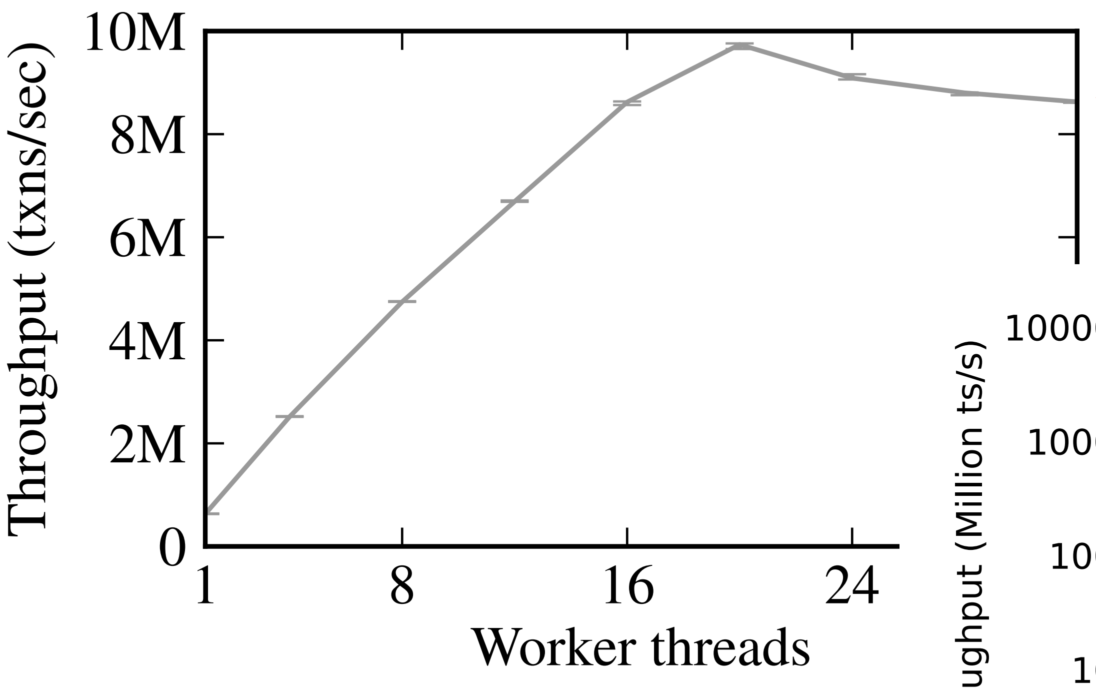
Scalability of Logging

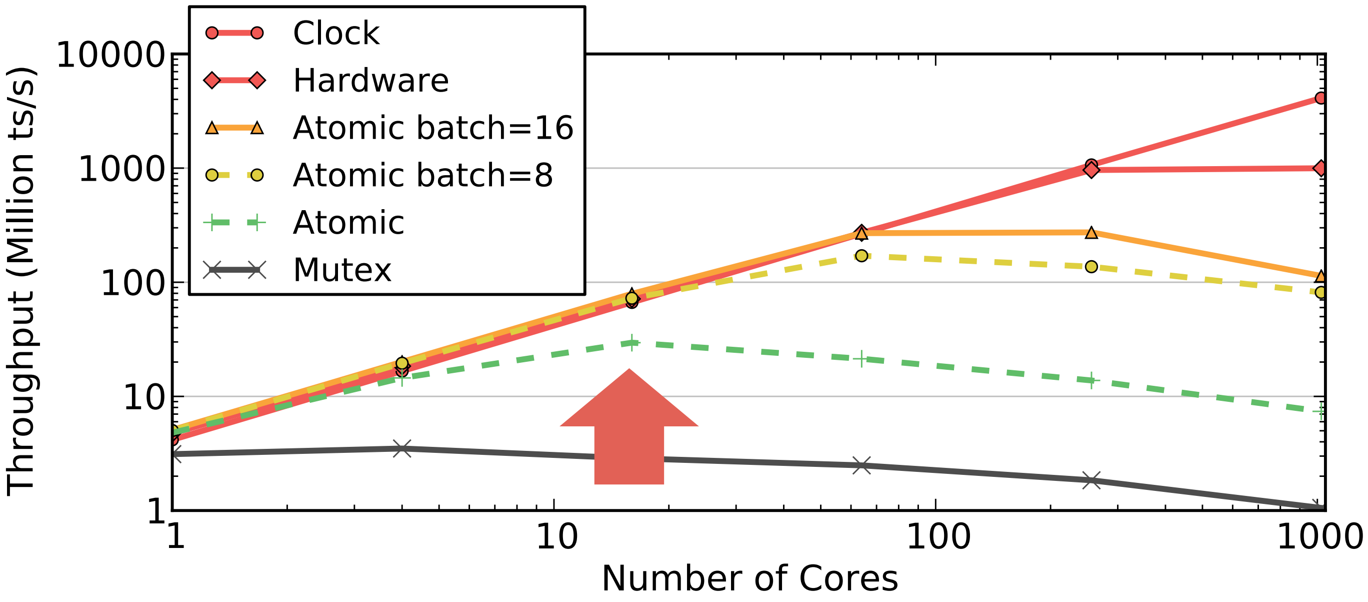
```
atomic_fetch_and_add(&lsn, size);
```



Scalability of Logging

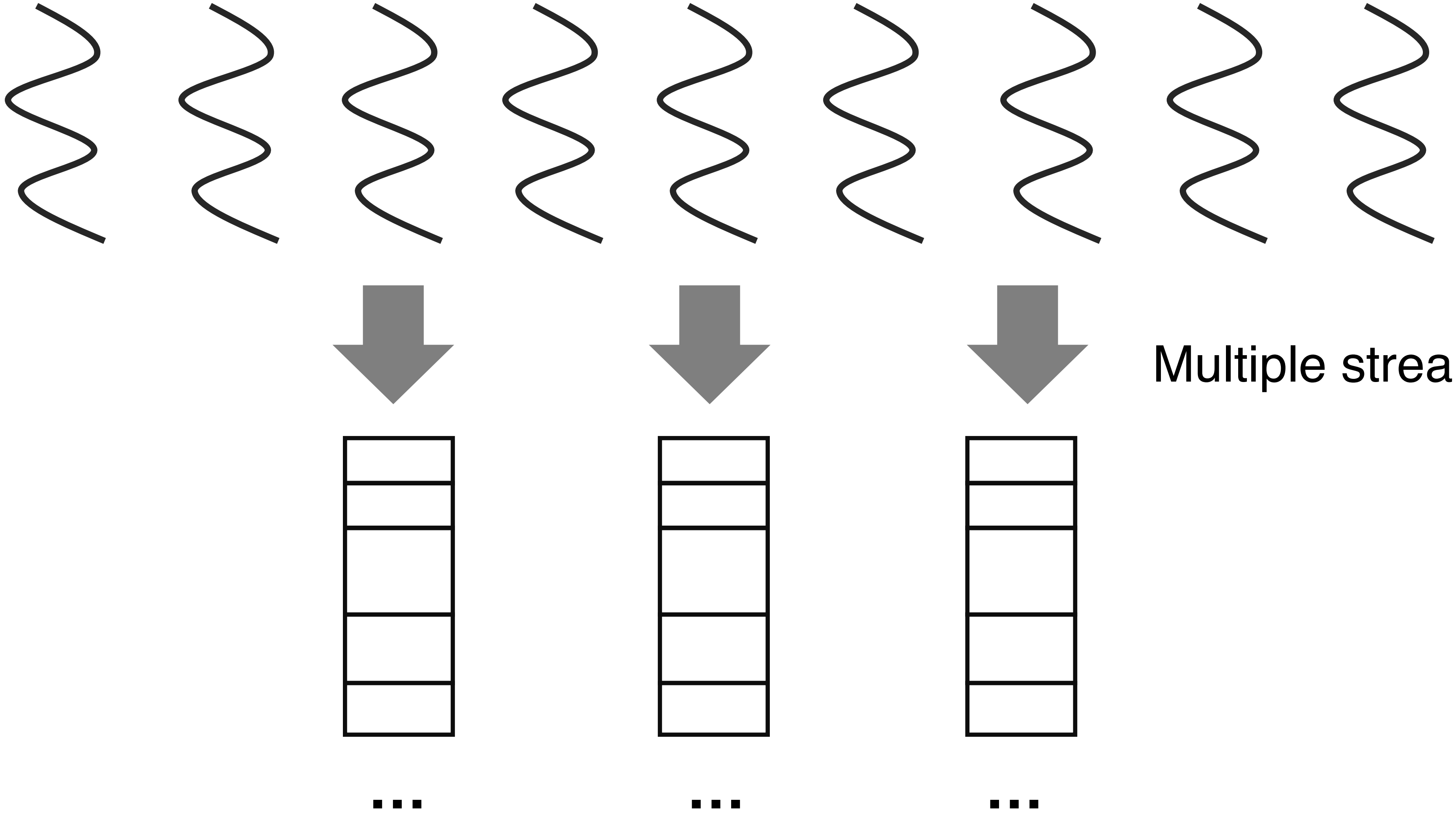
```
atomic_fetch_and_add(&lsn, size);
```





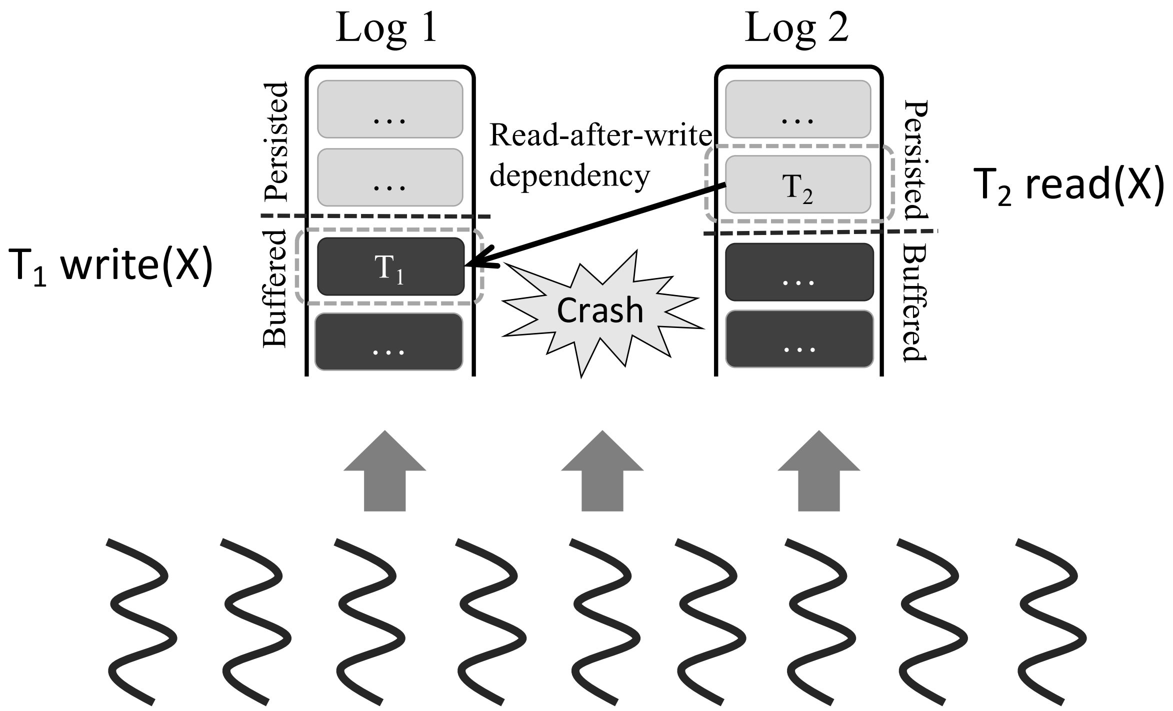
Why Have Global Transaction ID?

Why Have Global Transaction ID?



Multiple streams of logging

Challenges of Parallel Logging

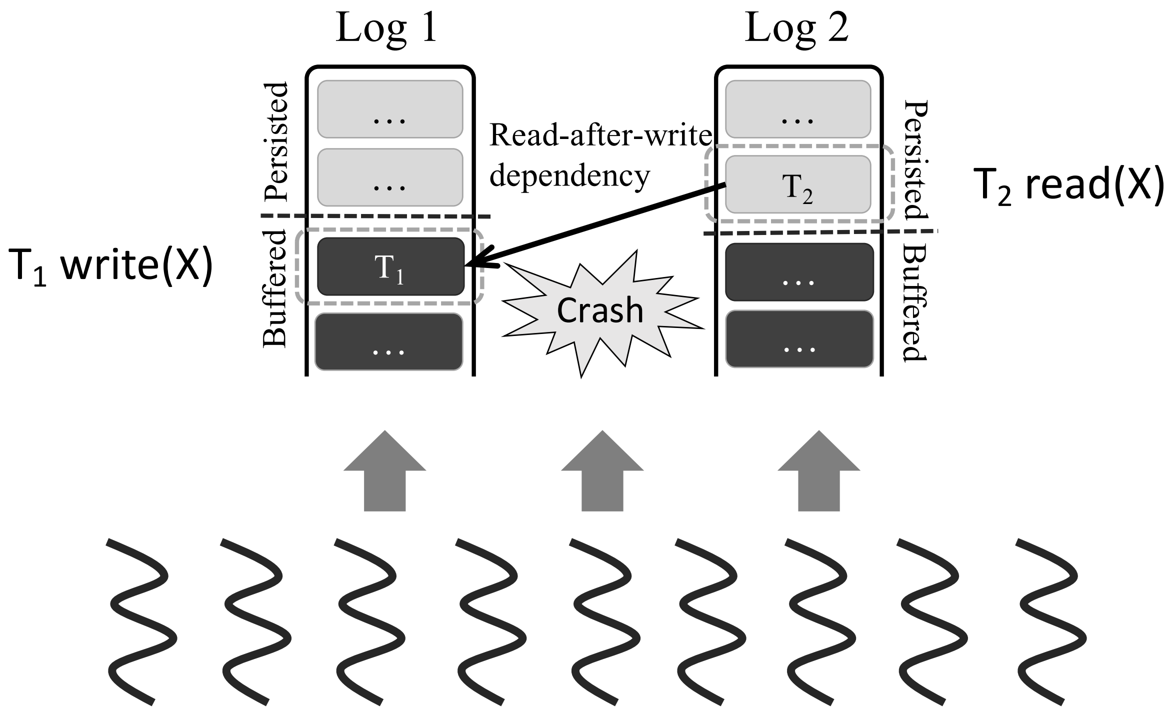


Challenge 1: When to commit?

Challenge 2: Whether to recover?

Challenge 3: How to recover?

Challenges of Parallel Logging



Challenge 1: When to commit?

- **Epoch-based commit**

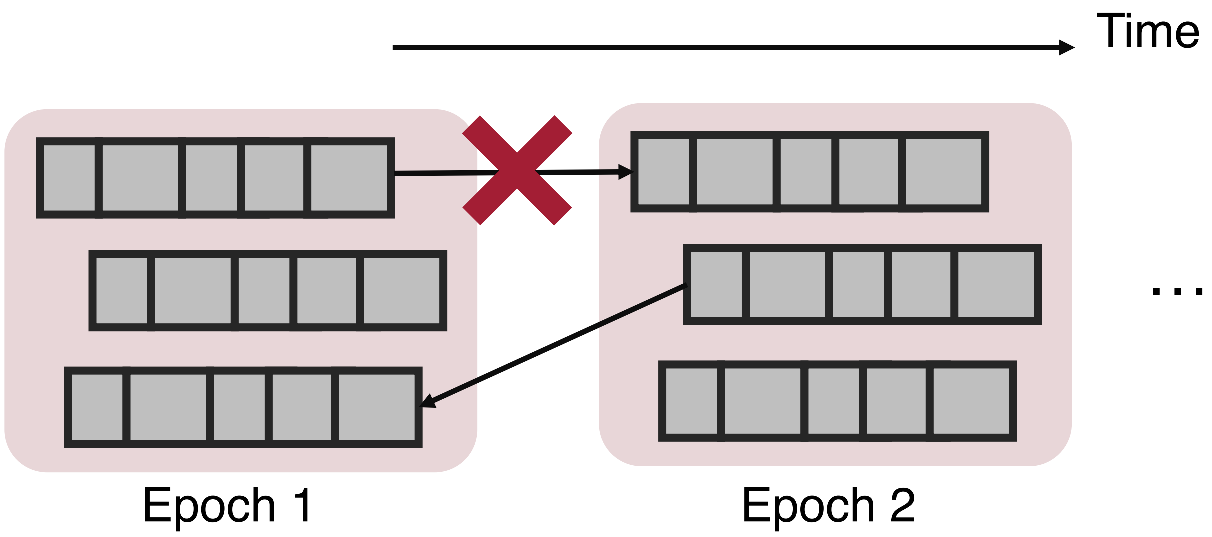
Challenge 2: Whether to recover?

- **Up to last complete epoch**

Challenge 3: How to recover?

- **Value-based logging/recovery**

Epoch-Based Design



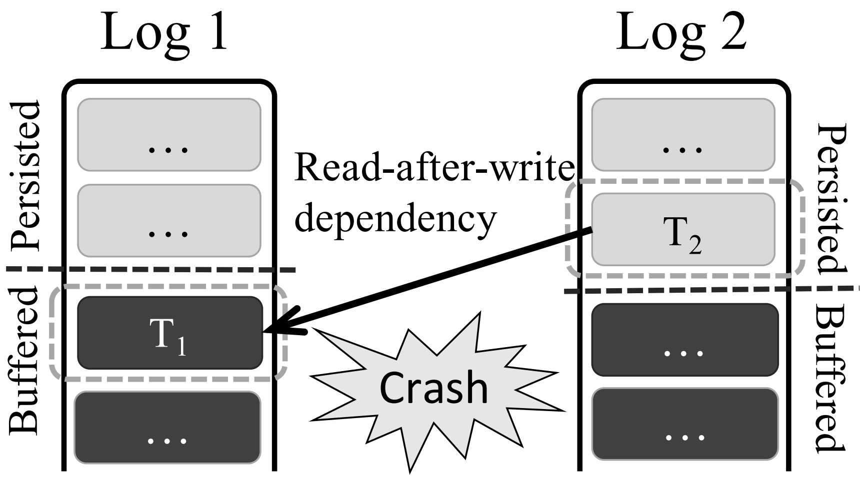
Challenge 1: When to commit?

- **Epoch-based commit**

Challenge 2: Whether to recover?

- **Up to last complete epoch**

Value-Based Logging



Challenge 3: How to recover?

- **Value-based logging/recovery**

No need to maintain read-after-write (RAW) or write-after-read (WAR) dependencies

Read-after-write (RAW): Flow dependency

Write-after-read (WAR): Anti dependency

Write-after-write (WAW): Output dependency

What about operational logging?

Epoch-Based OCC

```
// validation phase
validation():
    lock all records in WS
    e = get_epoch_number()
    for r in RS ∪ WS:
        if r.version != DB[r.key].version or r.is_locked:
            abort()
    version = gen_version(WS, RS, e)

// write phase
write():
    for r in WS:
        DB[r.key].value = r.value
        DB[r.key].version = version
        unlock(r)
```

Epoch read = serialization point

Key property: **epoch differences agree with dependencies**.

- T2 reads T1's write → T2's epoch ≥ T1's (T2 RAW depends on T1)
- T1 does not read T2'write → T2's epoch ≥ T1's (T2 WAR depends on T1)

Read-After-Write (RAW) Example

Key property: **epoch differences agree with dependencies**

- T2 reads T1's write → T2's epoch ≥ T1's (T2 RAW depends on T1)

Time

T1:

```
WriteLocal(A, 2);

Lock(A);
e = Global_Epoch;

t = GenerateTID(e);

WriteAndUnlock(A, t);
```

T2:

```
tmp = Read(A);
WriteLocal(A, tmp+1);

Lock(A);
e = Global_Epoch;

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

WriteAndUnlock(A, t);
```

T2's epoch ≥ T1's epoch
T2's TID > T1's TID

Read-After-Write (RAW) Example

Key property: **epoch differences agree with dependencies**

- T2 reads T1's write → T2's epoch ≥ T1's (T2 RAW depends on T1)

Time

T1:
WriteLocal(A, 2);

Lock(A);
e = Global_Epoch;

t = GenerateTID(e);

WriteAndUnlock(A, t);

T2:
tmp = Read(A);
WriteLocal(A, tmp+1);

Lock(A);
e = Global_Epoch;

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

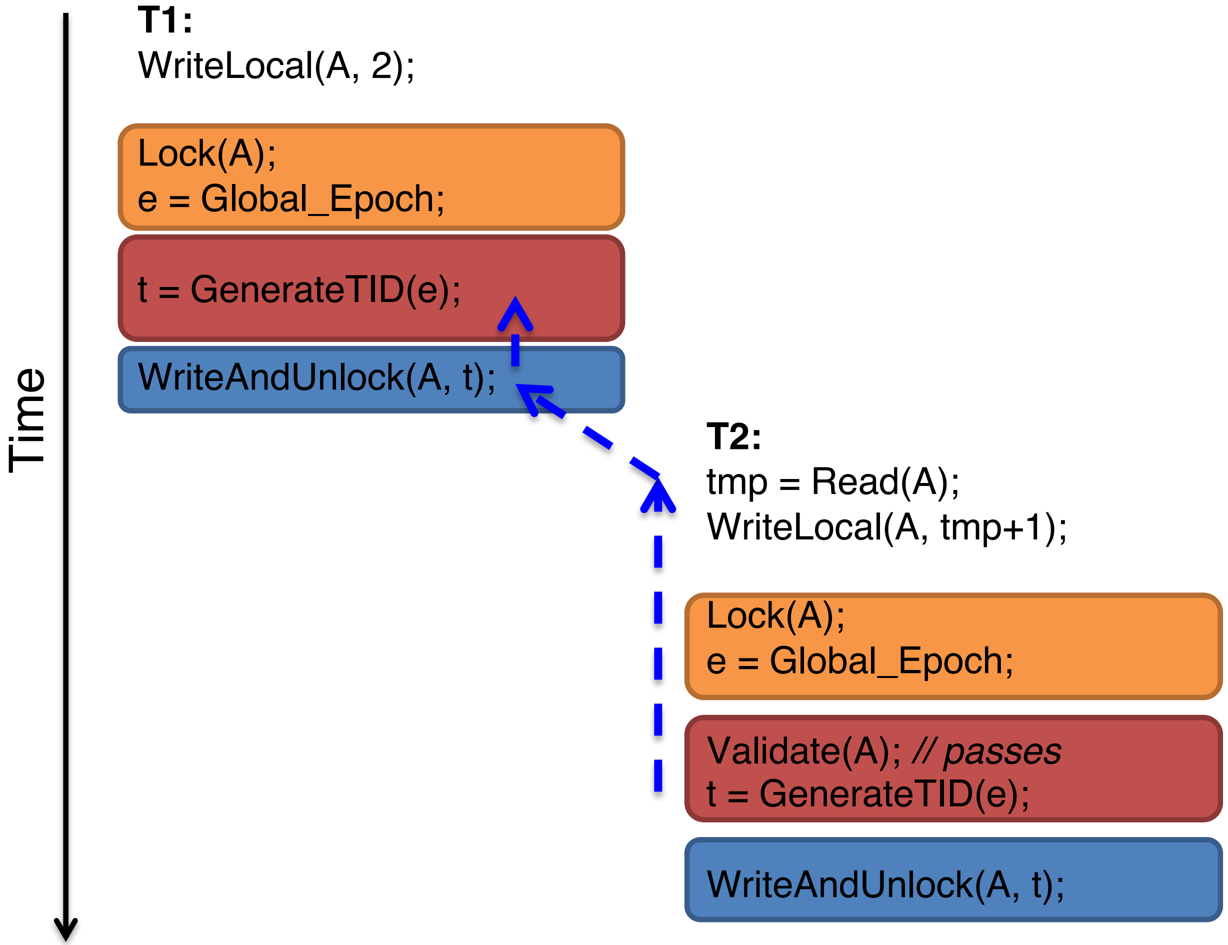
WriteAndUnlock(A, t);

T2's epoch ≥ T1's epoch

Read-After-Write (RAW) Example

Key property: **epoch differences agree with dependencies**

- T2 reads T1's write → T2's epoch ≥ T1's (T2 RAW depends on T1)



T2's epoch ≥ T1's epoch
T2's TID > T1's TID

Write-After-Read Example

Key property: **epoch differences agree with dependencies**

- T1 does not read T2'write → T2's epoch ≥ T1's (T2 WAR depends on T1)

Time

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

Lock(B);
e = Global_Epoch;

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

WriteAndUnlock(B, t);
```

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

Lock(A);
e = Global_Epoch;

t = GenerateTID(e);

WriteAndUnlock(A, t);
```

Write-After-Read Example

Key property: **epoch differences agree with dependencies**

- T1 does not read T2'write → T2's epoch ≥ T1's (T2 WAR depends on T1)

Time

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

Lock(B);
e = Global_Epoch;

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

WriteAndUnlock(B, t);
```

T2's epoch ≥ T1's epoch

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

Lock(A);
e = Global_Epoch;

t = GenerateTID(e);

WriteAndUnlock(A, t);
```

Low-Level Optimizations

Transaction Identifiers

Each record contains a TID word which is broken into three pieces:

Status bits	Sequence number	Epoch number
0		63

Status bits = A ***lock*** bit, a ***latest-version*** bit, and an ***absent*** bit

Assign TID at commit time (after reads).

- (a) larger than the TID of any record read or written by the transaction
- (b) larger than the worker’s most re- cently chosen TID
- (c) in the current global epoch. The result is written into each record modified by the transaction.

Read/Write

Status bits	Sequence number	Epoch number

0 63

```
// read phase
read(): read record into read set (RS)
```

Read/Write

Status bits	Sequence number	Epoch number

0 63

```
// read phase
read(): read record into read set (RS)
```

How to consistently read a record and its TID word?

Read/Write

Status bits	Sequence number	Epoch number

0 … 63

```
// read phase
read(): read record into read set (RS)
```

How to consistently read a record and its TID word?

```
// read record t
do
    v1 = t.read_TID_word()
    RS[t.key].data = t.data
    v2 = t.read_TID_word()
while (v1 != v2 or v1.lock_bit == 1);
```

Range Scan and Phantom

Phantom reads:

New rows are added or removed by another transaction to the records being read

Perform the scan twice

Ok if the second scan returns the same set of record

Silo Commit Protocol

```
// Validation phase
for w, v in sorted(WS)
    Lock(w); // use a lock bit in TID

Fence(); // compiler-only on x86
e = Global_Epoch; // serialization point
Fence(); // compiler-only on x86

for r, t in RS
    Validate(r, t); // abort if fails

tid = Generate_TID(RS, WS, e);

// Write phase
for w, v in WS {
    Write(w, v, tid);
    Unlock(w);
}
```

Experimental Evaluation

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

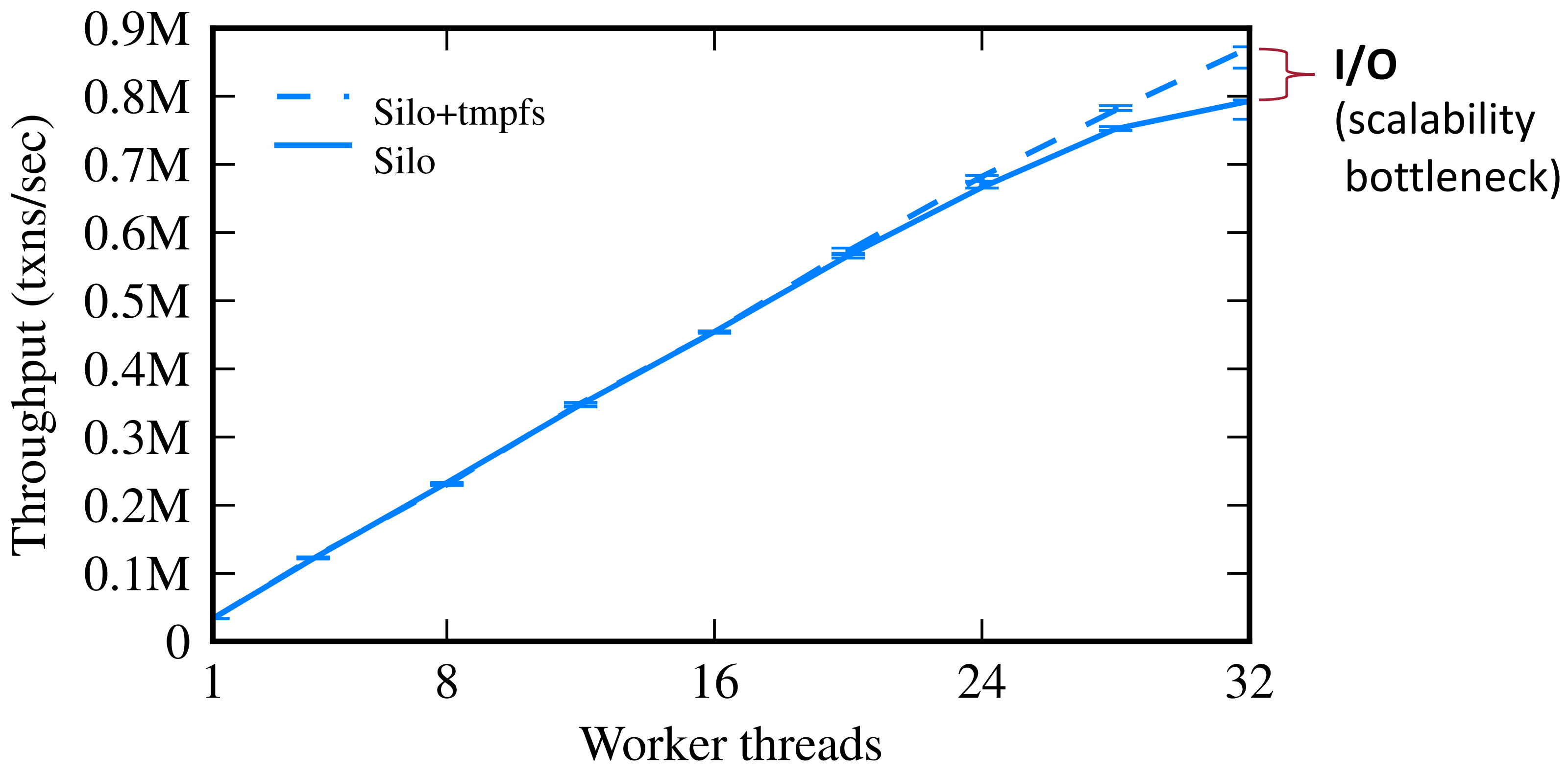
TPC-C

- Average log record length is ~1KB
- All loggers combined writing ~1GB/sec

YCSB

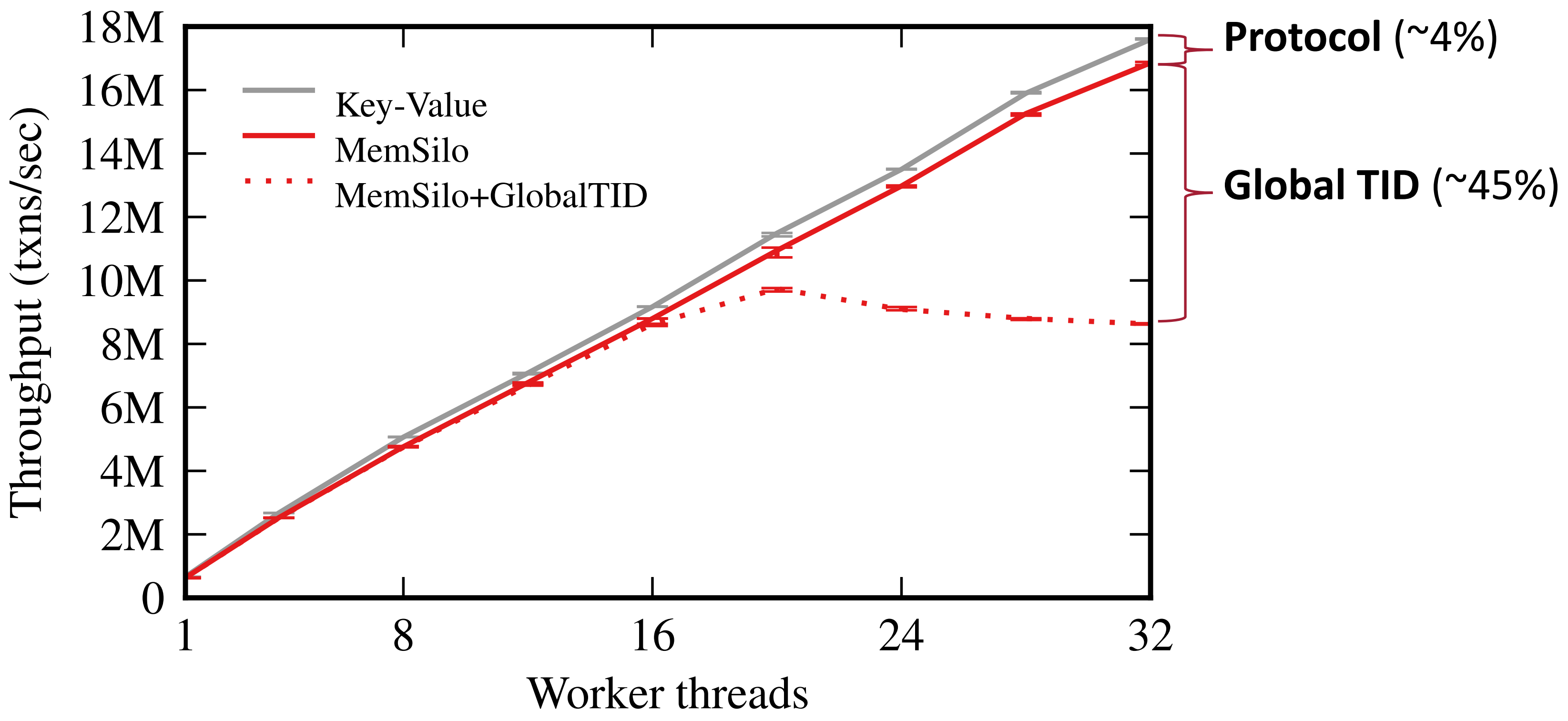
- 80/20 read/read-modify-write
- 100 byte records
- Uniform key distribution

Silo on TPC-C



I/O slightly limits scalability, protocol does not.

Silo on YCSB



Key-Value: Masstree (no multi-key transactions).

- Transactional commits are inexpensive.

MemSilo+GlobalTID: A single compare-and-swap added to commit protocol.

Summary

Even a single atomic_add instruction can limit scalability

Silo remove bottleneck through epochs

When to commit? **Epoch-based commit**

Whether to recover? **Up to last complete epoch**

How to recover? **Value-based logging/recovery**

Low-level optimizations for high performance

- TID word
- Invisible reads

Silo – Q/A

Limitations:

- Only one-shot transactions (Good enough for real systems?)
- Value-based logging
- Long transactions
- Long latency

Adoptions of epochs

- Many follow-up research papers use epochs
- Deterministic DB (next lecture) uses epochs

What is a memory/compiler fence?

Group Discussion

Is Silo compatible with operational logging?

- Operational logging: log the operations instead of the values. The DB re-executes the transactions during recovery

One downside of Silo is long transaction latency (due to epochs), can you come up with any solution to this problem?

What are the challenges of applying Silo to a distributed database?

Before Next Lecture

Submit discussion summary to https://wisc-cs839-ngdb20.hotcrp.com

- **Deadline: Wednesday 11:59pm**

Submit review for

Calvin: Fast Distributed Transactions for Partitioned Database Systems

[optional] Rethinking serializable multiversion concurrency control (Extended Version)

[optional] An Evaluation of Distributed Concurrency Control