CS 764: Topics in Database Management Systems

Lecture 17: Optimistic Concurrency Control

Xiangyao Yu

10/30/2025

Today's Paper: Optimistic Concurrency Control

On Optimistic Methods for Concurrency Control

H.T. KUNG and JOHN T. ROBINSON
Carnegie-Mellon University

Most current approaches to concurrency control in database systems rely on locking of data objects as a control mechanism. In this paper, two families of nonlocking concurrency controls are presented. The methods used are "optimistic" in the sense that they rely mainly on transaction backup as a control mechanism, "hoping" that conflicts between transactions will not occur. Applications for which these methods should be more efficient than locking are discussed.

Key Words and Phrases: databases, concurrency controls, transaction processing
CR Categories: 4.32, 4.33

1. INTRODUCTION

Consider the problem of providing shared access to a database organized as a collection of objects. We assume that certain distinguished objects, called the roots, are always present and access to any object other than a root is gained only by first accessing a root and then following pointers to that object. Any sequence of accesses to the database that preserves the integrity constraints of the data is called a *transaction* (see, e.g., [4]).

If our goal is to maximize the throughput of accesses to the database, then there are at least two cases where highly concurrent access is desirable.

ACM Trans. Database Syst. 1981

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.

1 Introduction

Thanks to drastic increases in main memory sizes and processor core counts for server-class machines, modern high-end servers can have several terabytes of RAM and 80 or more cores. When used effectively, this is enough processing power and memory to handle data sets and computations that used to be spread across many disks and machines. However, harnassing this power is tricky; even single points of contention, like compare-and-swaps on a shared-memory word, can limit scalability.

This paper presents Silo, a new main-memory database that achieves excellent performance on multicore machines. We designed Silo from the ground up to use system memory and caches efficiently. We avoid all centralized contention points and make all synchronization 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 serial order. Our core result, the Silo commit protocol, is a minimal-contention serializable commit protocol that provides these properties.

Silo uses a variant of optimistic concurrency control (OCC) [18]. An OCC transaction tracks the records it reads and writes in thread-local storage. At commit time, after validating that no concurrent transaction's writes overlapped with its read set, the transaction installs all written records at once. If validation fails, the transaction aborts. This approach has several benefits for scalability. OCC writes to shared memory only at commit time, after the transaction's compute phase has completed; this short write period reduces contention. And thanks to the validation step, read-set records need not be locked. This matters because the memory writes required for read locks can induce contention [11].

Previous OCC implementations are not free of scaling bottlenecks, however, with a key reason being the requirement for tracking "anti-dependencies" (write-after-read conflicts). Consider a transaction t_1 that reads a record from the database, and a concurrent transaction t_2 that overwrites the value t_1 saw. A serializable system must order t_1 before t_2 even after a potential crash and recovery from persistent logs. To achieve this ordering, most systems require that t_1 communicate with t_2, such as by posting its read sets to shared memory or via a centrally-assigned, monotonically-increasing transaction ID [18, 19]. Some non-serializable systems can avoid this communication, but they suffer from anomalies like snapshot isolation's "write skew" [2].

Silo provides serializability while avoiding *all* shared-memory writes for read transactions. The commit protocol was carefully designed using memory fences to scalably produce results consistent with a serial order. This leaves the problem of correct recovery, which we solve using a form of *epoch-based group commit*. Time is divided into a series of short epochs. Even though transaction results always agree with a serial order, the system

SOSP'13, Nov. 3–6, 2013, Farmington, Pennsylvania, USA.
ACM 978-1-4503-2388-8/13/11.
http://dx.doi.org/10.1145/2517349.2522713

SOSP, 2013

Agenda

Downsides of pessimistic concurrency control

Optimistic concurrency control

- Read phase
- Write phase
- Validation phase

Modern OCC: Silo

Concurrency Control

Concurrency control ensures the correctness for concurrent operations

Assume **serializable** isolation level for this lecture

Concurrency Control

Concurrency control ensures the correctness for concurrent operations

Assume **serializable** isolation level for this lecture

Pessimistic: Resolve conflicts eagerly

Optimistic: Ignore conflicts during a transaction's execution and resolve conflicts lazily only when at a transaction's completion time

Concurrency Control

Concurrency control ensures the correctness for concurrent operations

Assume **serializable** isolation level for this lecture

Pessimistic: Resolve conflicts eagerly

Optimistic: Ignore conflicts during a transaction's execution and resolve conflicts lazily only when at a transaction's completion time

Other common concurrency control protocols

- Timestamp ordering (T/O)
- Multi-version concurrency control (MVCC)

Pessimistic Concurrency Control

Strict two-phase locking (2PL)

- Acquire the right type of locks before accessing data
- Release locks when the transaction commits

Pessimistic Concurrency Control

Strict two-phase locking (2PL)

- Acquire the right type of locks before accessing data
- Release locks when the transaction commits

Downsides of pessimistic concurrency control

- Locking overhead, even for read-only transactions
- Deadlocks
- Limited concurrency due to (1) congestion and (2) holding locks till the end of a transaction

Pessimistic Concurrency Control

Strict two-phase locking (2PL)

- Acquire the right type of locks before accessing data
- Release locks when the transaction commits

Downsides of pessimistic concurrency control

- Locking overhead, even for read-only transactions
- Deadlocks
- Limited concurrency due to (1) congestion and (2) holding locks till the end of a transaction

Observation: Locking is needed only if contention exists; real workloads have low contention

Agenda

Downsides of pessimistic concurrency control

Optimistic concurrency control

- Read phase
- Write phase
- Validation phase

Modern OCC: Silo

Optimistic Concurrency Control (OCC)

Goal: eliminating pessimistic locking

Three executing phases:

- Read
- Validation
- Write

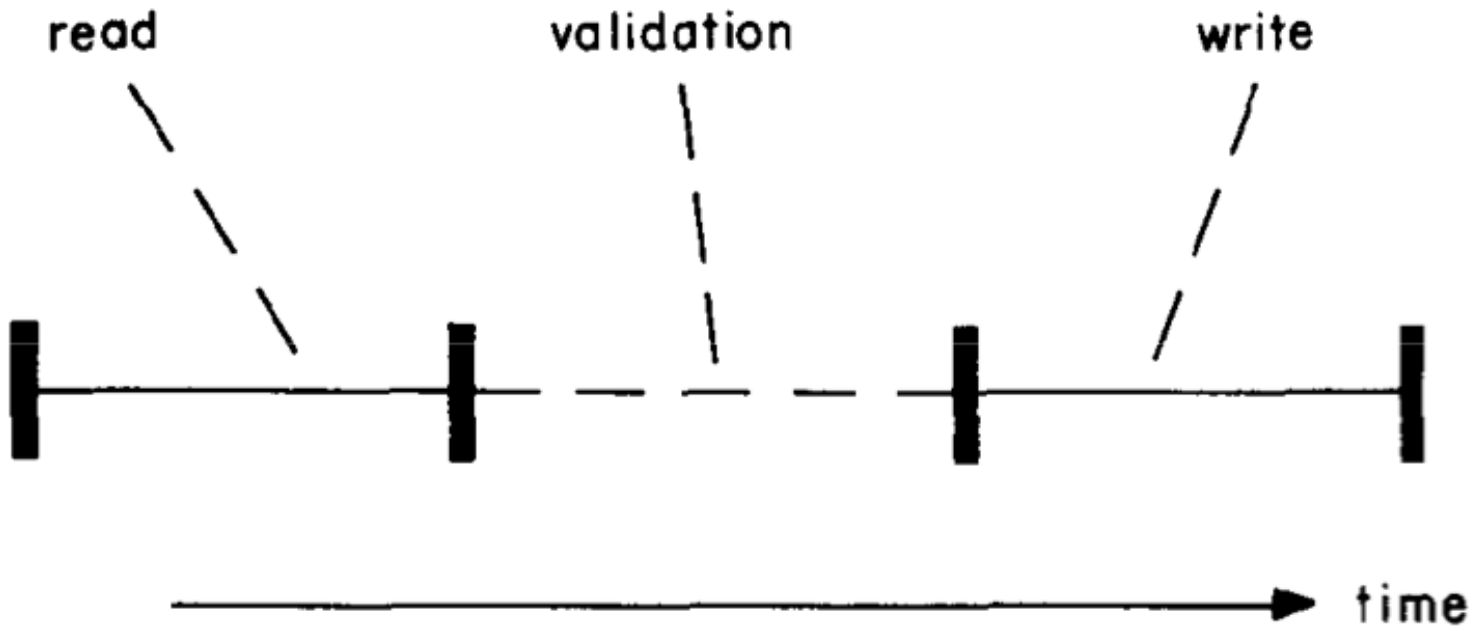


Fig. 1. The three phases of a transaction.

Read Phase

n = *tcreate*

- Insert a new record

$$
\begin{aligned}
&tcreate = (\\
&\quad n := create;\\
&\quad create\ set := create\ set \cup \{n\};\\
&\quad \textbf{return}\ n)
\end{aligned}
$$

Read Phase

n = tcreate

twrite(n, i, v)

- Write to local write set
- No modification to the database

```
twrite(n, i, v) = (
   if n ∈ create set
        then write(n, i, v)
   else if n ∈ write set
        then write(copies[n], i, v)
   else (
        m := copy(n);
        copies[n] := m;
        write set := write set ∪ {n};
        write(copies[n], i, v)))
```

Read Phase

n = tcreate

twrite(n, i, v)

value = tread(n, i)

- Read from either the local write set or the database

```
tread(n, i) = (
    read set := read set ∪ {n};
    if n ∈ write set
        then return read(copies[n], i)
    else
        return read(n, i))
```

Read Phase

n = tcreate

twrite(n, i, v)

value = tread(n, i)

tdelete(n)

- Mark delete in local delete set
- No deletion from the database

$$tdelete(n) = ($$
$$\quad delete\ set := delete\ set \cup \{n\}).$$

Read Phase

n = tcreate

twrite(n, i, v)

value = tread(n, i)

tdelete(n)

All changes (i.e., inserts, updates, deletes) are kept local to the transaction without updating the database

Write Phase

All written values become “global”

$$\textbf{for } n \in \mathit{write\ set} \textbf{ do } \mathit{exchange}(n, \mathit{copies}[n]).$$

All created nodes become accessible

All deleted nodes become inaccessible

Validation Phase

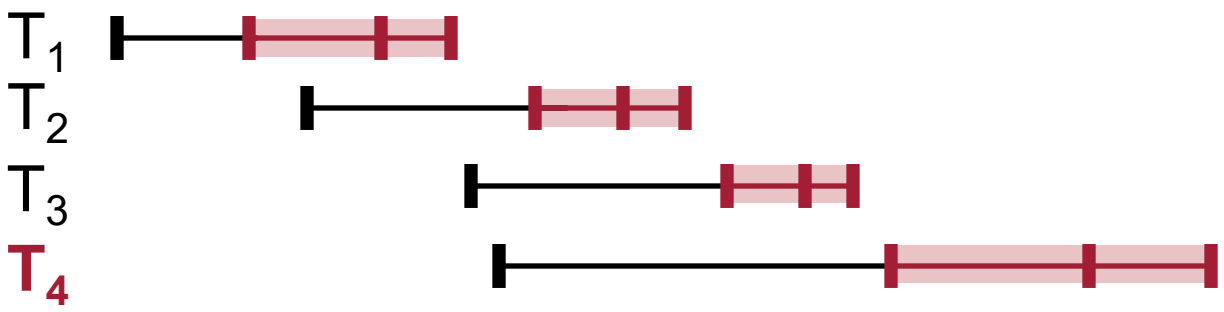
A transaction *i* is assigned a transaction number *t(i)* when it enters the validation phase

- Transaction number determines global serialization order
- *t(i)* < *t(j)* => exists a serial schedule where T_i is before T_j
- If execution does not obey this order, the validating transaction aborts

Serial Validation

```
tbegin = (
    start tn := tnc)
tend = (
  ⟨finish tn := tnc;
   valid := true;
   for t from start tn + 1 to finish tn do
        if (write set of transaction with transaction number t intersects read set)
           then valid := false;
   if valid
        then ((write phase); tnc := tnc + 1; tn := tnc)⟩;
   if valid
        then (cleanup)
        else (backup)).
```

Critical Section

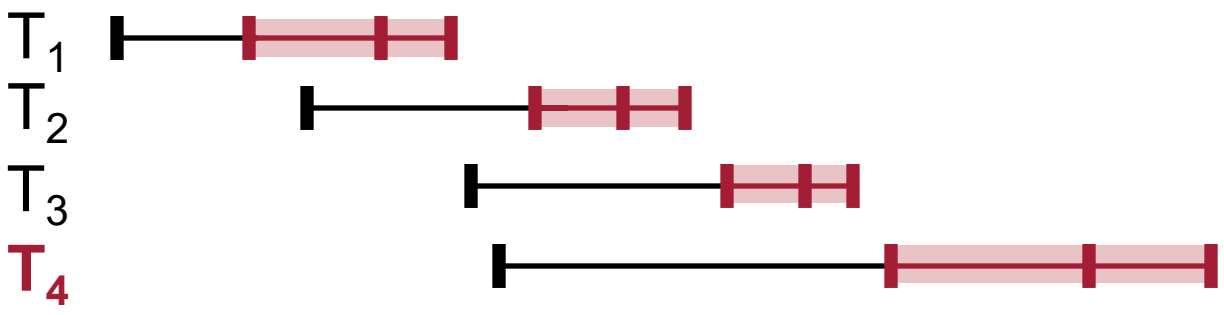


Serial Validation

```
tbegin = (
    start tn := tnc)
tend = (
  ⟨finish tn := tnc;
   valid := true;
   for t from start tn + 1 to finish tn do
        if (write set of transaction with transaction number t intersects read set)
           then valid := false;
   if valid
        then ((write phase); tnc := tnc + 1; tn := tnc)⟩;
   if valid
        then (cleanup)
        else (backup)).
```

Critical Section

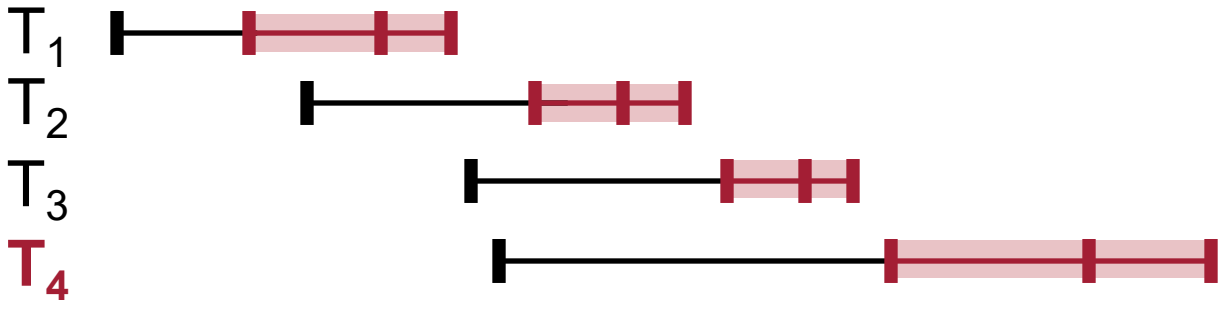
Which transactions will T2, T3, and T4 be validated against?



Serial Validation

```
tbegin = (
    start tn := tnc)
tend = (
  ⟨finish tn := tnc;
   valid := true;
   for t from start tn + 1 to finish tn do
        if (write set of transaction with transaction number t intersects read set)
           then valid := false;
   if valid
        then ((write phase); tnc := tnc + 1; tn := tnc)⟩;
   if valid
        then (cleanup)
        else (backup)).
```

Critical Section



Which transactions will T2, T3, and T4 be validated against?

Problem: Both *validate* and *write* phases happen in the critical section

Improved Serial Validation

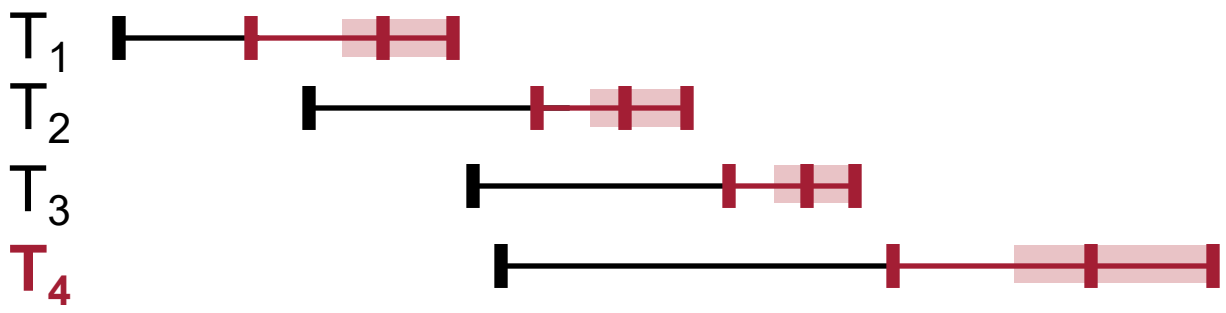
```
tend := (
    mid tn := tnc;
    valid := true;
    for t from start tn + 1 to mid tn do
    if (write set of transaction with transaction number t intersects read set)
        then valid := false;
⟨finish tn := tnc;
 for t from mid tn + 1 to finish tn do
        if (write set of transaction with transaction number t intersects read set)
          then valid := false;
 if valid
        then ((write phase); tnc := tnc + 1; tn := tnc)⟩;
    if valid
        then (cleanup)
        else (backup)).
```

Critical Section

Part of the validation process happens outside the critical section

The optimization can be applied repeatedly

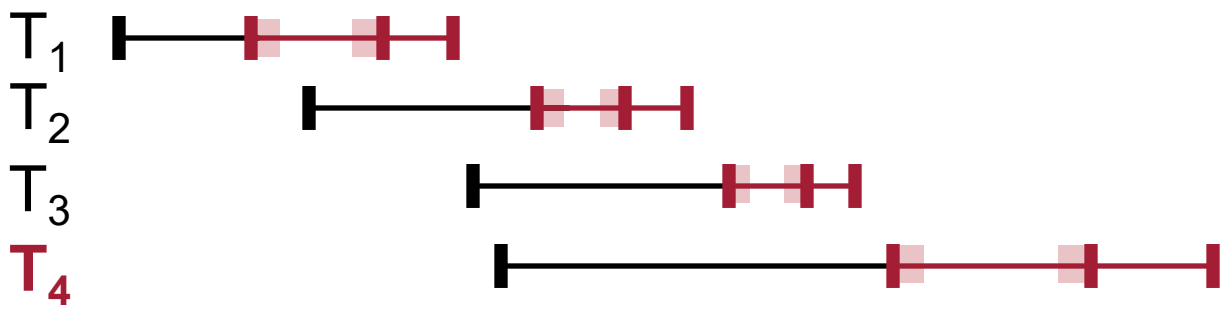
Readonly transactions do not enter the critical section



Parallel Validation

```
tend = (
⟨finish tn := tnc;
 finish active := (make a copy of active);
 active := active ∪ {id of this transaction}⟩;
 valid := true;
 for t from start tn + 1 to finish tn do
      if (write set of transaction with transaction number t intersects read set)
        then valid := false;
 for i ∈ finish active do
      if (write set of transaction Ti intersects read set or write set)
        then valid := false;
 if valid
      then (
        (write phase);
        ⟨tnc := tnc + 1;
         tn := tnc;
         active := active—{id of this transaction}⟩;
        (cleanup))
      else (
        ⟨active := active—{id of transaction}⟩;
        (backup))).
```

Critical Sections



Validation against other transactions and writes both happen outside the critical section

Length of the critical section is independent of the number of validating transactions

Leading to unnecessary aborts

- Abort due to conflict with an aborted transaction

Parallel Validation

```
tend = (
  ⟨finish tn := tnc;
   finish active := (make a copy of active);
   active := active ∪ {id of this transaction}⟩;
   valid := true;
   for t from start tn + 1 to finish tn do
       if (write set of transaction with transaction number t intersects read set)
         then valid := false;
   for i ∈ finish active do
       if (write set of transaction T_i intersects read set or write set
         then valid := false;
   if valid
       then (
         (write phase);
         ⟨tnc := tnc + 1;
          tn := tnc;
          active := active—{id of this transaction}⟩;
         (cleanup))
       else (
         ⟨active := active—{id of transaction}⟩;
         (backup))).
```

Question: Why need to consider both read set and write set when validating against transactions in *finish active*?

Agenda

Downsides of pessimistic concurrency control

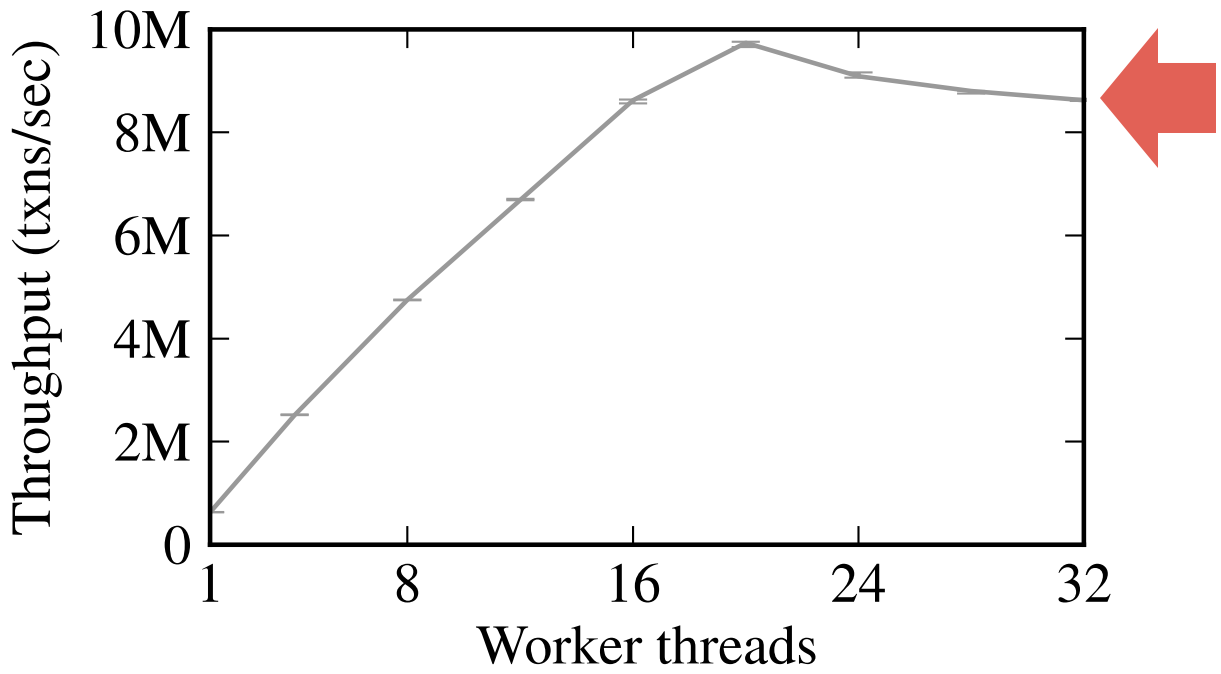
Optimistic concurrency control

- Read phase
- Write phase
- Validation phase

Modern OCC: Silo

Timestamp Allocation Bottleneck

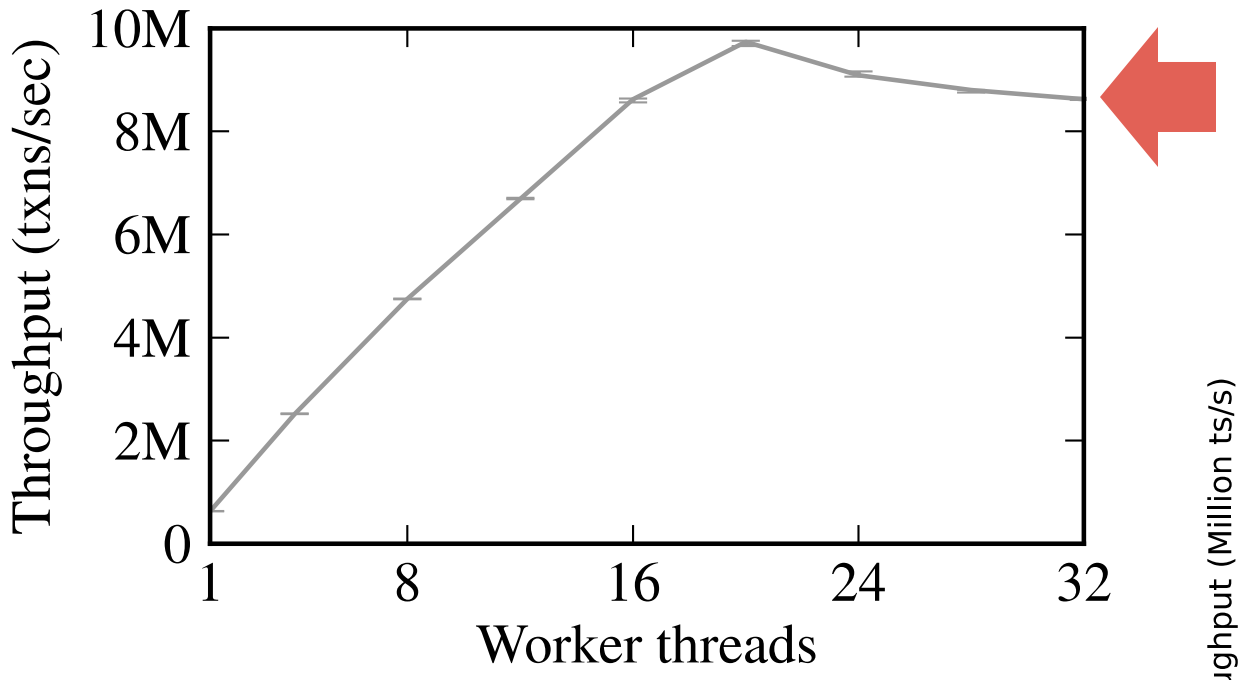
```
atomic_fetch_and_add(&lsn, size);
```



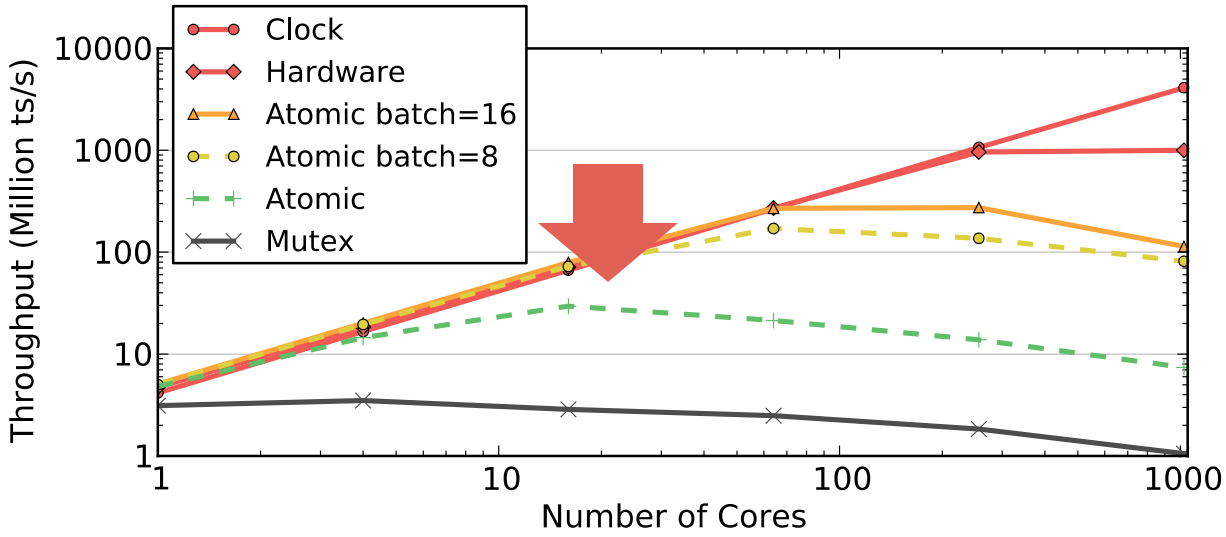
Even a single atomic instruction can become a scalability bottleneck

Timestamp Allocation Bottleneck

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



Even a single atomic instruction can become a scalability bottleneck



X. Yu et al. *Staring into the Abyss: An Evaluation of Concurrency Control with One Thousand Cores*, VLDB 2014

Silo Read Phase

Each tuple contains a 64-bit TID word

Silo Read Phase

Each tuple contains a 64-bit TID word

Each read returns consistent **value** and **TID word**

- Method 1: Guard the read with a latch (i.e., a short lock)
- Method 2: Optimistic lock (Silo's approach)

Silo Read Phase

Each tuple contains a 64-bit TID word

Each read returns consistent **value** and **TID word**

- Method 1: Guard the read with a latch (i.e., a short lock)
- Method 2: Optimistic lock (Silo's approach)

```
// read a record
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);
```

Silo Read Phase

Each tuple contains a 64-bit TID word

Each read returns consistent **value** and **TID word**

- Method 1: Guard the read with a latch (i.e., a short lock)
- Method 2: Optimistic lock (Silo's approach)

```
// read a record
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);
```

```
// write a record
v1.lock_bit = 1
v1.update()
v1.update_seq_number()
v1.lock_bit = 0
```

Silo Validation Phase

```
Data: read set R, write set W, node set N,
      global epoch number E
// Phase 1
for record, new-value in sorted(W) do
    lock(record);
compiler-fence();
e ← E;                              // serialization point
compiler-fence();
// Phase 2
for record, read-tid in R do
    if record.tid ≠ read-tid or not record.latest
        or (record.locked and record ∉ W)
    then abort();
for node, version in N do
    if node.version ≠ version then abort();
commit-tid ← generate-tid(R, W, e);
// Phase 3
for record, new-value in W do
    write(record, new-value, commit-tid);
    unlock(record);
```

Phase 1: Lock the write set

Silo Validation Phase

```
Data: read set R, write set W, node set N,
      global epoch number E
// Phase 1
for record, new-value in sorted(W) do
    lock(record);
compiler-fence();
e ← E;                                  // serialization point
compiler-fence();
// Phase 2
for record, read-tid in R do
    if record.tid ≠ read-tid or not record.latest
          or (record.locked and record ∉ W)
    then abort();
for node, version in N do
    if node.version ≠ version then abort();
commit-tid ← generate-tid(R, W, e);
// Phase 3
for record, new-value in W do
    write(record, new-value, commit-tid);
    unlock(record);
```

Phase 1: Lock the write set

Q: Why need to sort write set?

Silo Validation Phase

```
Data: read set R, write set W, node set N,
      global epoch number E
// Phase 1
for record, new-value in sorted(W) do
    lock(record);
compiler-fence();
e ← E;                                  // serialization point
compiler-fence();
// Phase 2
for record, read-tid in R do
    if record.tid ≠ read-tid or not record.latest
        or (record.locked and record ∉ W)
    then abort();
for node, version in N do
    if node.version ≠ version then abort();
commit-tid ← generate-tid(R, W, e);
// Phase 3
for record, new-value in W do
    write(record, new-value, commit-tid);
    unlock(record);
```

Phase 1: Lock the write set

Phase 2: Validate the read set

- Validation fails if (1) the tuple is modified since the earlier read or (2) the tuple is locked by another transaction

Silo Validation Phase

```
Data: read set R, write set W, node set N,
      global epoch number E
// Phase 1
for record, new-value in sorted(W) do
    lock(record);
compiler-fence();
e ← E;                                  // serialization point
compiler-fence();
// Phase 2
for record, read-tid in R do
    if record.tid ≠ read-tid or not record.latest
        or (record.locked and record ∉ W)
    then abort();
for node, version in N do
    if node.version ≠ version then abort();
commit-tid ← generate-tid(R, W, e);
// Phase 3
for record, new-value in W do
    write(record, new-value, commit-tid);
    unlock(record);
```

Phase 1: Lock the write set

Phase 2: Validate the read set

- Validation fails if (1) the tuple is modified since the earlier read or (2) the tuple is locked by another transaction

Q: If a tuple is modified since a transaction's earlier read, can the transaction still be serializable?

Silo Validation Phase

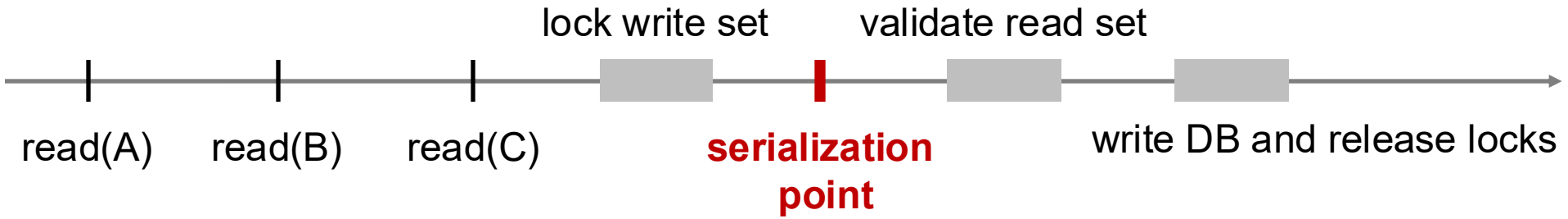
```
Data: read set R, write set W, node set N,
      global epoch number E
// Phase 1
for record, new-value in sorted(W) do
    lock(record);
compiler-fence();
e ← E;                                  // serialization point
compiler-fence();
// Phase 2
for record, read-tid in R do
    if record.tid ≠ read-tid or not record.latest
          or (record.locked and record ∉ W)
    then abort();
for node, version in N do
    if node.version ≠ version then abort();
commit-tid ← generate-tid(R, W, e);
// Phase 3
for record, new-value in W do
    write(record, new-value, commit-tid);
    unlock(record);
```

Phase 1: Lock the write set

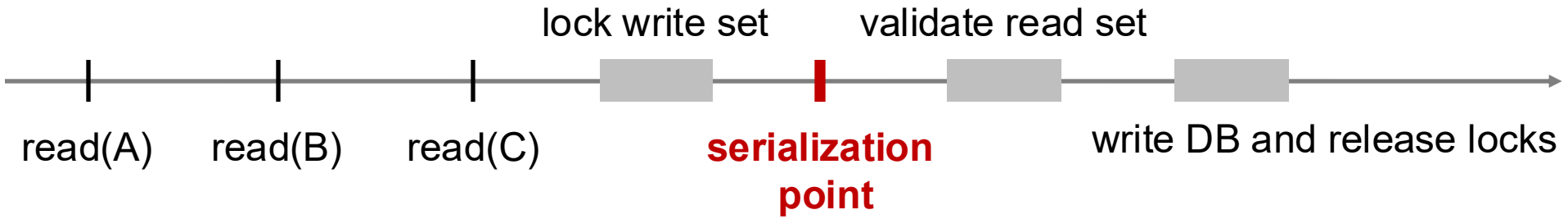
Phase 2: Validate the read set

Phase 3: Write phase

Silo OCC is Serializable



Silo OCC is Serializable



Proof idea

- The Silo schedule is equivalent to an idealized schedule where all reads and writes of a transaction occur at the serialization point
- (Same strategy can be used to prove that 2PL is serializable)

Silo vs. OCC 1981

Silo

```
// Phase 1
for record, new-value in sorted(W) do
    lock(record);
compiler-fence();
e ← E;                                  // serialization point
compiler-fence();
// Phase 2
for record, read-tid in R do
    if record.tid ≠ read-tid or not record.latest
        or (record.locked and record ∉ W)
    then abort();
```

OCC 1981

```
tend = (
  ⟨finish tn := tnc;
   valid := true;
   for t from start tn + 1 to finish tn do
       if (write set of transaction with transaction number t intersects read set)
         then valid := false;
   if valid
       then ((write phase); tnc := tnc + 1; tn := tnc)⟩;
   if valid
       then (cleanup)
       else (backup)).
```

Silo vs. OCC 1981

Silo

```
// Phase 1
for record, new-value in sorted(W) do
    lock(record);
compiler-fence();
e ← E;                                  // serialization point
compiler-fence();
// Phase 2
for record, read-tid in R do
    if record.tid ≠ read-tid or not record.latest
        or (record.locked and record ∉ W)
    then abort();
```

OCC 1981

```
tend = (
  ⟨finish tn := tnc;
   valid := true;
   for t from start tn + 1 to finish tn do
       if (write set of transaction with transaction number t intersects read set)
         then valid := false;
   if valid
       then ((write phase); tnc := tnc + 1; tn := tnc)⟩;
   if valid
       then (cleanup)
       else (backup)).
```

- Silo **locks tuples** in write set; OCC'81 uses **global critical sections**

Silo vs. OCC 1981

Silo

```
// Phase 1
for record, new-value in sorted(W) do
    lock(record);
compiler-fence();
e ← E;                                  // serialization point
compiler-fence();
// Phase 2
for record, read-tid in R do
    if record.tid ≠ read-tid or not record.latest
        or (record.locked and record ∉ W)
    then abort();
```

OCC 1981

```
tend = (
  ⟨finish tn := tnc;
   valid := true;
   for t from start tn + 1 to finish tn do
       if (write set of transaction with transaction number t intersects read set)
         then valid := false;
   if valid
       then ((write phase); tnc := tnc + 1; tn := tnc)⟩;
   if valid
       then (cleanup)
       else (backup)).
```

- Silo **locks tuples** in write set; OCC'81 uses **global critical sections**
- Silo validates using tuple versions; OCC'81 validates against write set of previous transactions

2PL vs. OCC

Revisit the motivation of OCC:

- **Locking overhead**, even for read-only transactions
- **Deadlocks**
- **Limited concurrency** due to (1) congestion and (2) holding locks till the end of a transaction

Comments:

- Optimized locks have low overhead, relative to disk and network cost
- When 2PL has limited concurrency, OCC may have high abort rate

Q/A – OCC

Modern approaches to solve starvation besides locking?

Long-running analytical transactions with massive read sets?

OCC compared to modern MVCC systems?

Choose between optimistic vs. pessimistic CC?

2PL vs. OCC under high contention?

OCC feasible in distributed system?

Hybrid OCC and 2PL?

Next Week—Midterm Exam