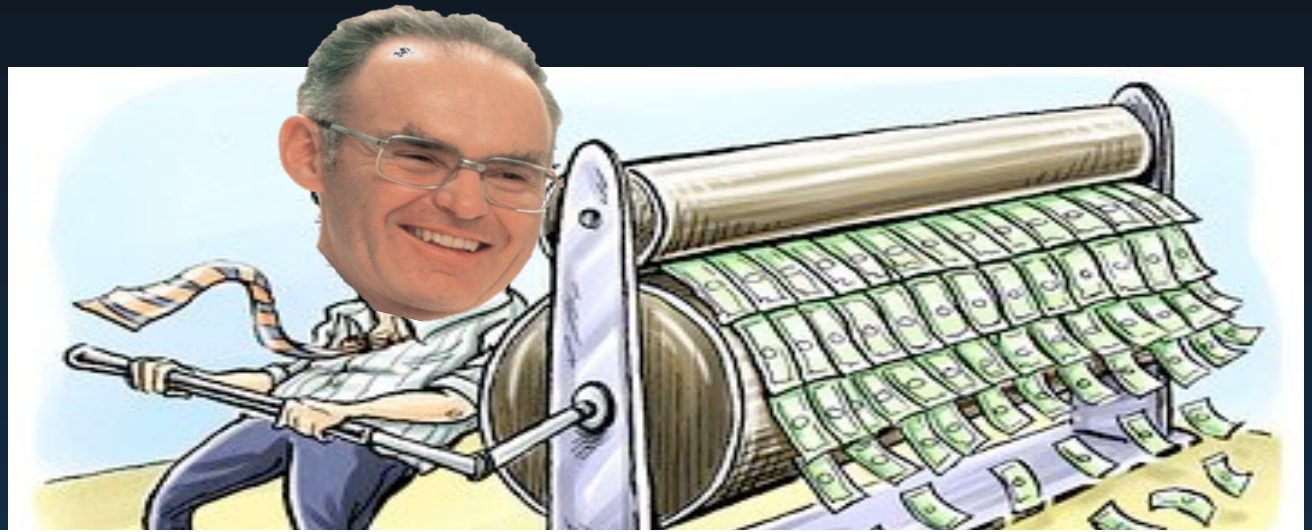
Time Traveling Hardware and Software Systems

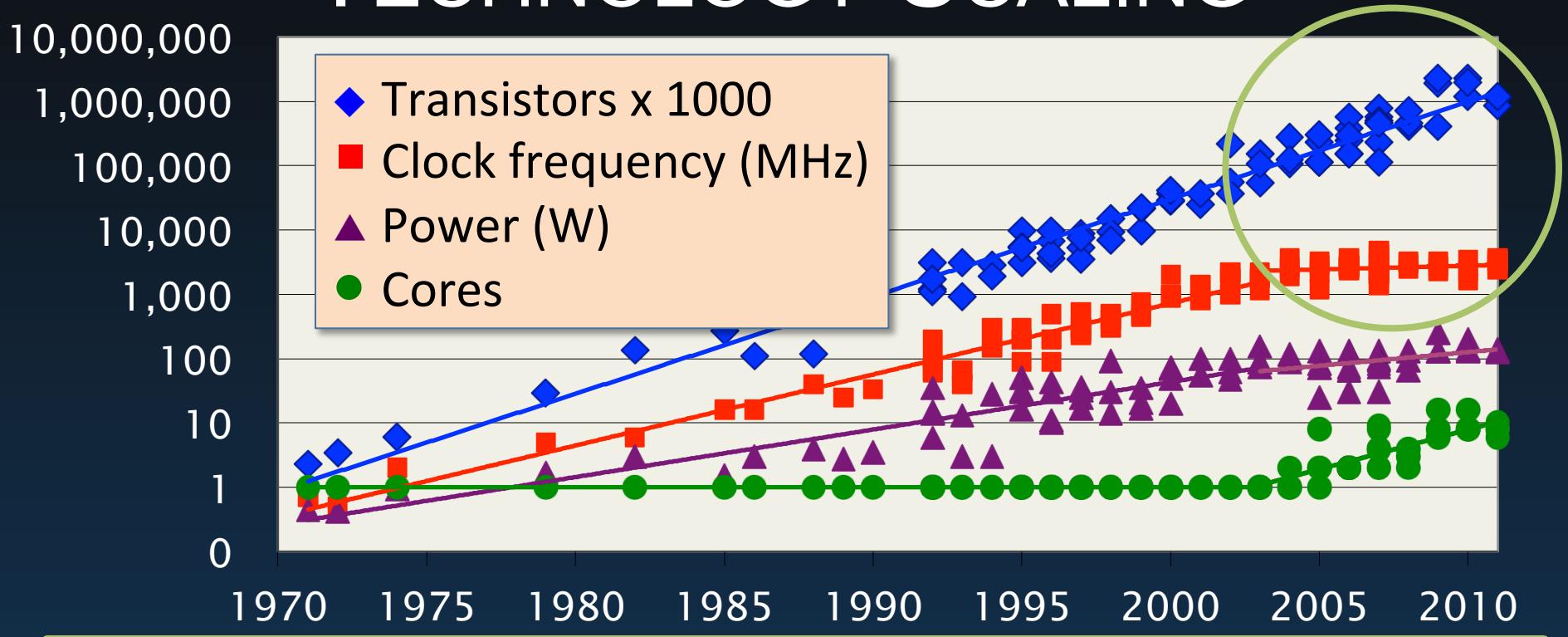
Xiangyao Yu, Srini Devadas
CSAIL, MIT

For Fifty years, we have ridden Moore's Law



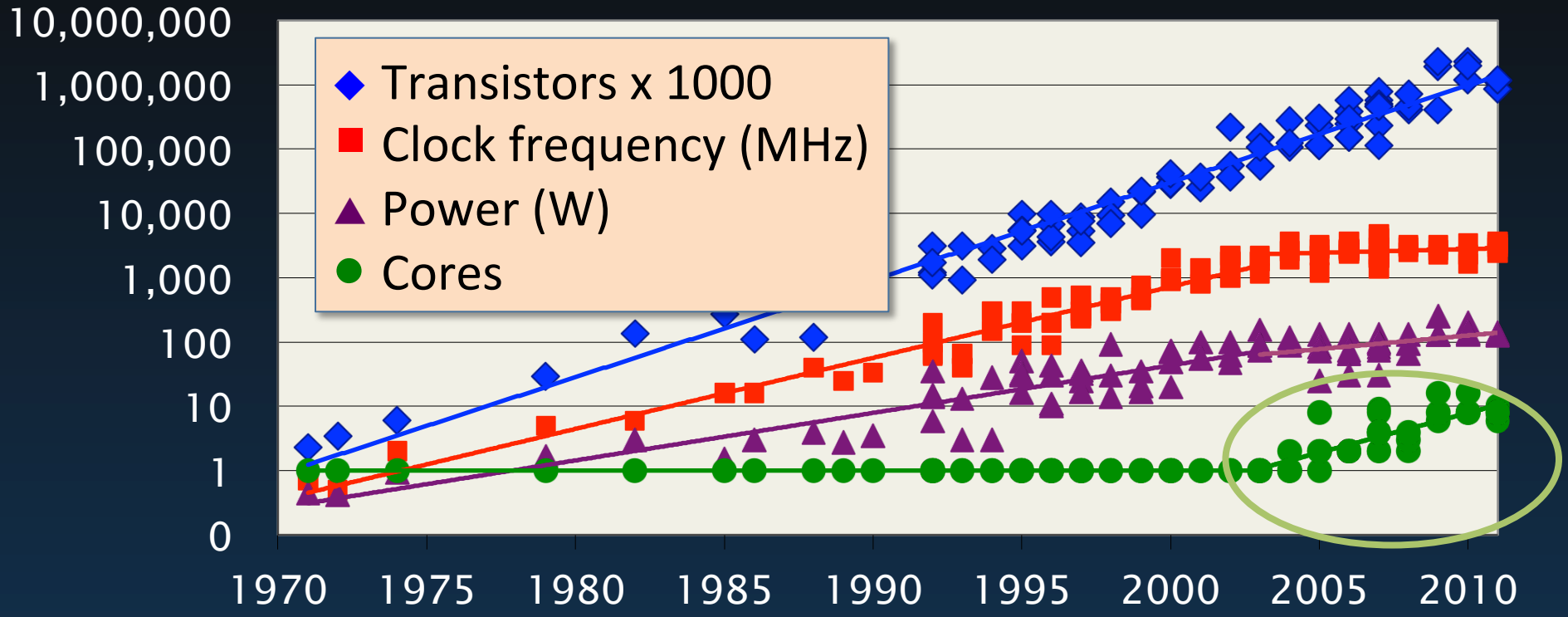
Moore's Law and the scaling of clock frequency
= printing press for the currency of performance

Technology Scaling



Each generation of Moore's law doubles the number of transistors but clock frequency has stopped increasing.

Technology Scaling



To increase performance, need to exploit parallelism.

Different Kinds of Parallelism - 1

Instruction Level

a = b + c
d = e + f
g = d + b

Transaction Level

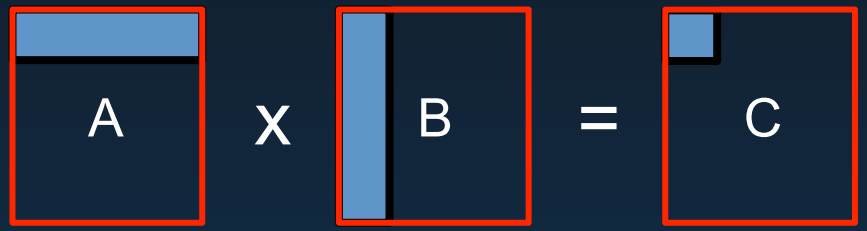
Read A
Read B
Compute C

Read A
Read D
Compute E

Read C
Read E
Compute F

Different Kinds of Parallelism - 2

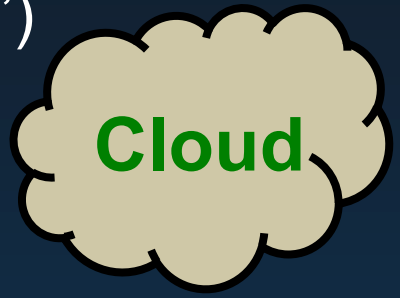
Thread Level



Different thread computes each entry of product matrix C

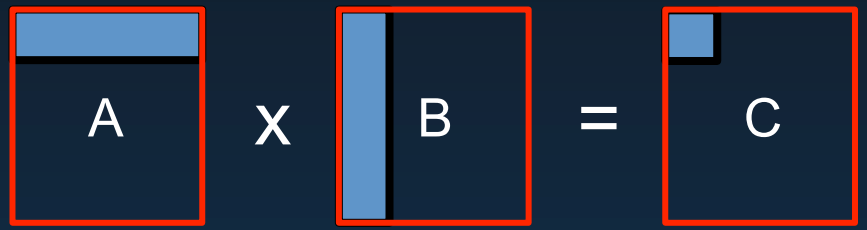
Task Level

Search(“image”)



Different Kinds of Parallelism - 3

Thread Level



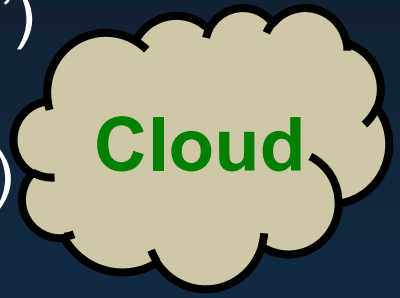
Different thread computes each entry of product matrix C

User Level

Search(“image”)

Lookup(“data”)

Query(“record”)



Dependency Destroys Parallelism

```
for i = 1 to n
    a[b[i]] = (a[b[i - 1]] + b[i]) / c[i]
```

Need to compute i^{th} entry after $i - 1^{th}$ has been computed ☹

Different Kinds of Dependency

Read A
Read A

No dependency!

Write A
Write A

WAW: Semantics decide order

Write A
Read A

RAW: Read needs new value

Read A
Write A

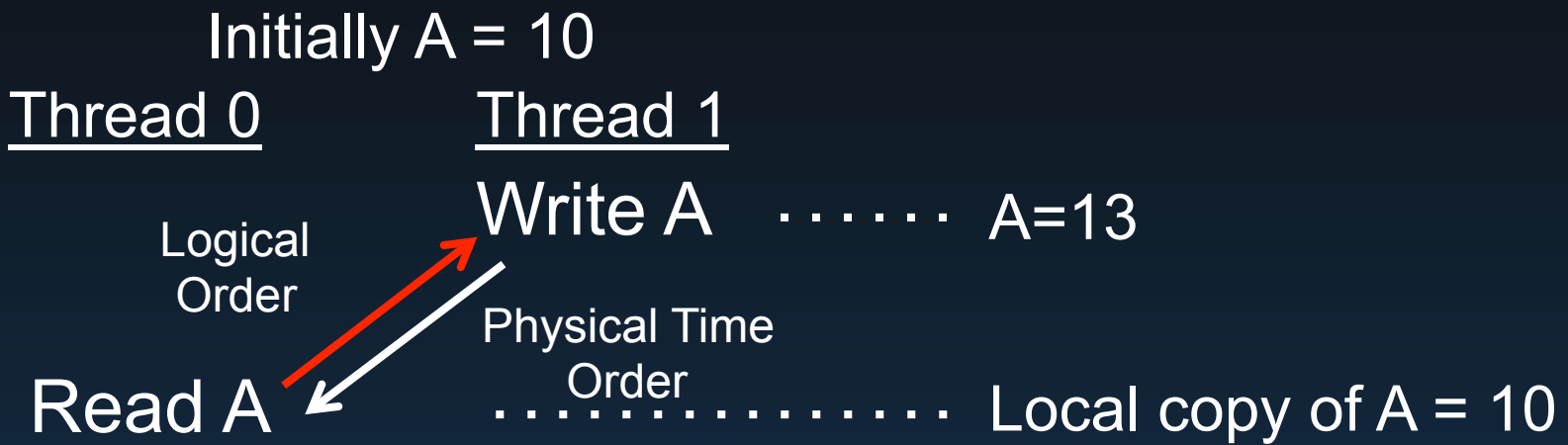
WAR: We have flexibility here!

Dependence is Across Time, But What Is Time?

- Time can be physical time
- Time could correspond to logical timestamps assigned to instructions
- Time could be a combination of the above

→ Time is a definition of ordering

WAR Dependence



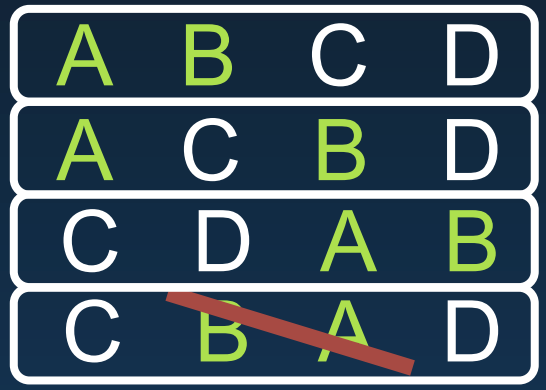
Read happens later than Write in physical time but is before Write in logical time.

What Is Correctness?

- We define correctness of a parallel program based on its outputs in relation to the program run sequentially

SEQUENTIAL CONSISTENCY

Global Memory Order



Can we exploit this freedom in correct execution to avoid dependency?

Avoiding Dependency Across the Stack

Circuit — Efficient atomic instructions

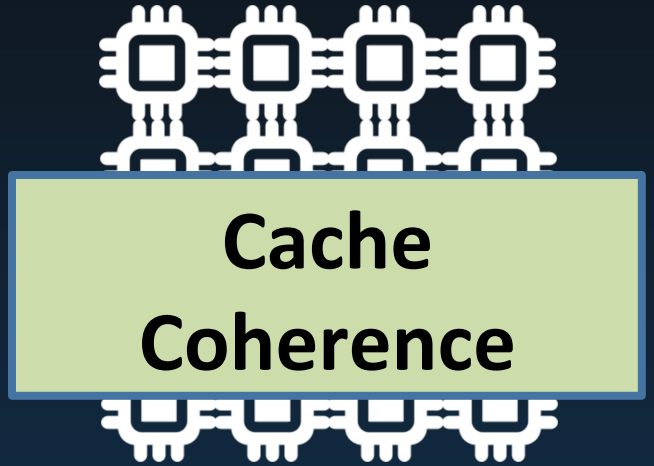
Multicore Processor — **Tardis coherence protocol**

Multicore Database — **TicToc concurrency control** ***with Andy Pavlo and Daniel Sanchez***

Distributed Database — Distributed TicToc

Distributed Shared Memory — Transaction processing with fault tolerance

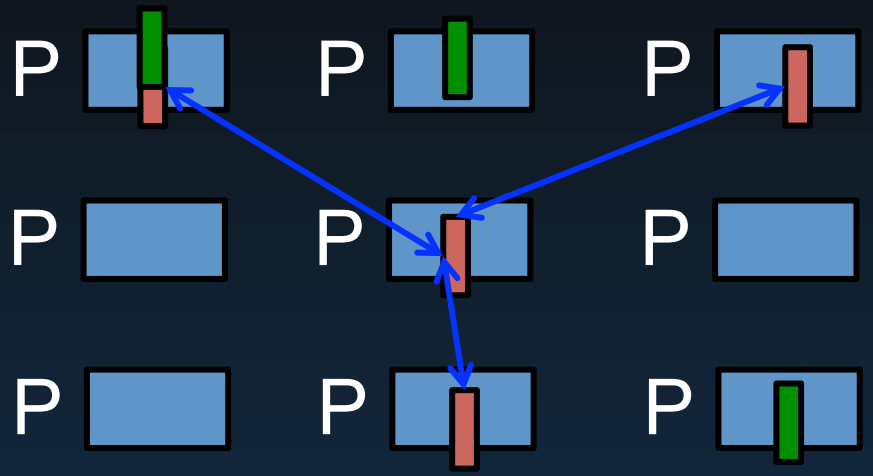
Shared Memory Systems



Multi-core
Processor

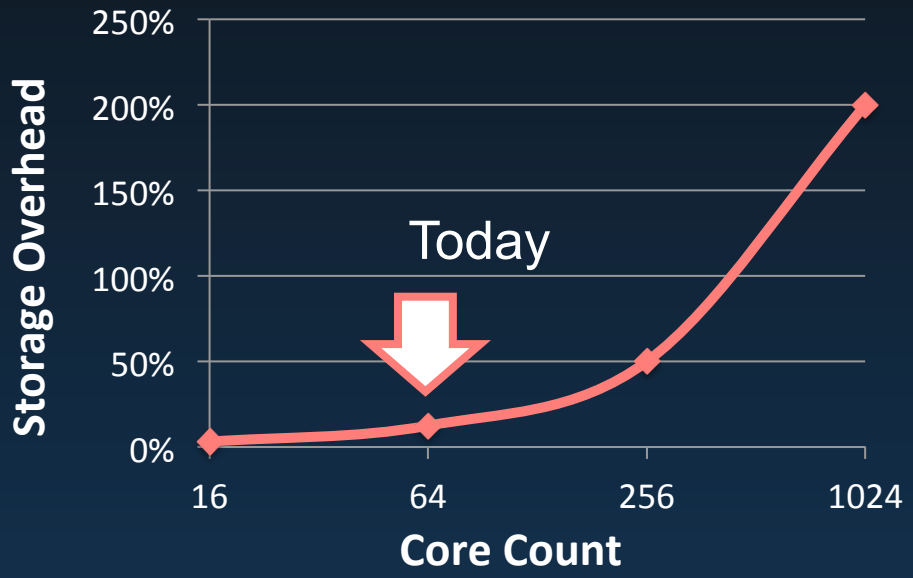
OLTP
Database

Directory-Based Coherence



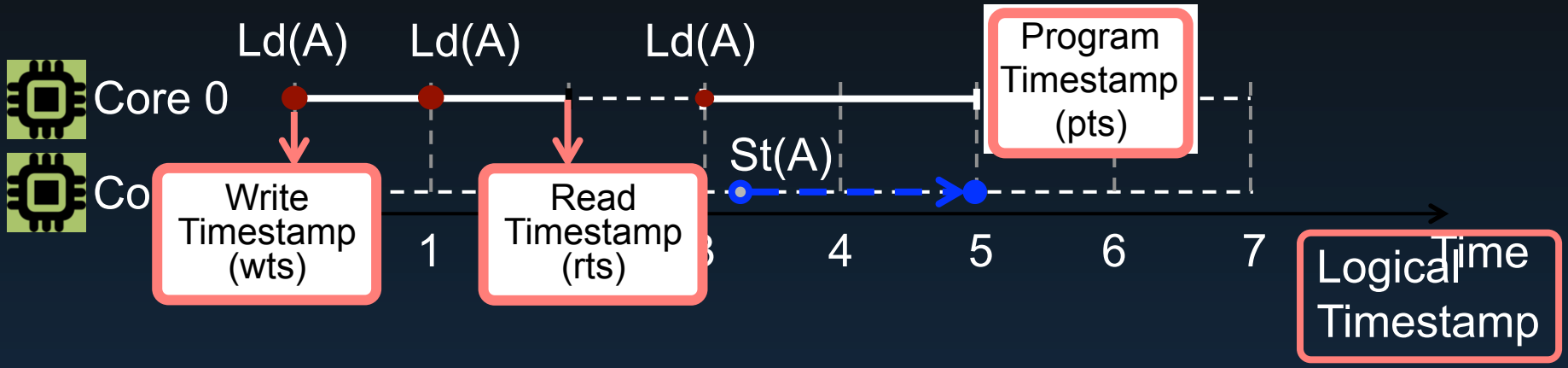
- Data replicated and cached locally for access
- Uncached data copied to local cache, writes invalidate data copies

Cache Coherence Scalability



O(N) Sharer List

LEASE-BASED COHERENCE



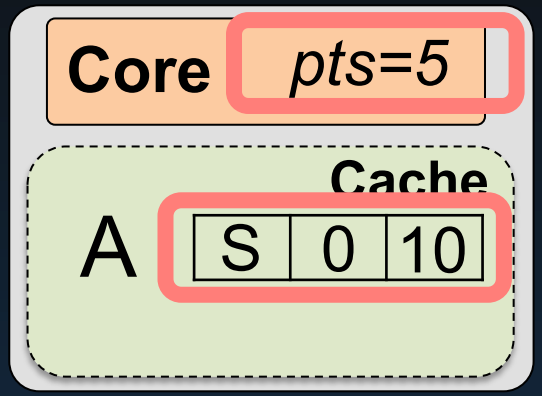
- A read gets a lease on a cacheline
- Lease renewal after lease expires
- A store can only commit after leases expire
- Tardis: logical leases

Logical Timestamp

Invalidation ⇐ Physical Time Order

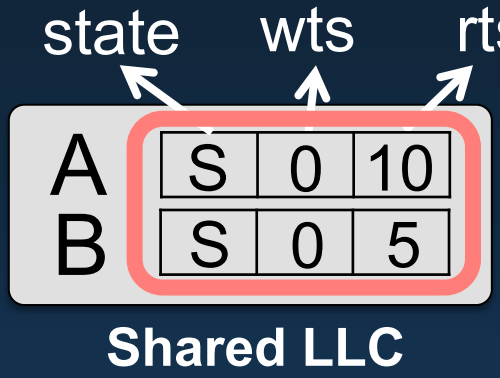
Tardis (No Invalidation) ⇐ Logical Time Order (concept borrowed from database)

Timestamp Management

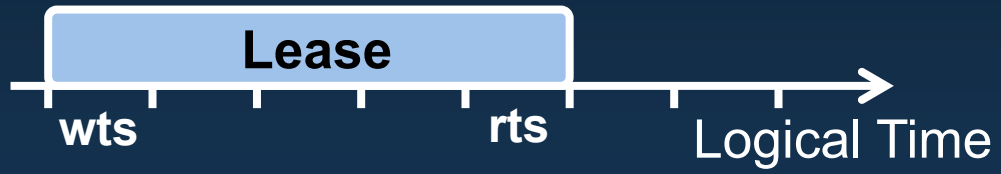


Program Timestamp (*pts*)
Timestamp of last memory operation

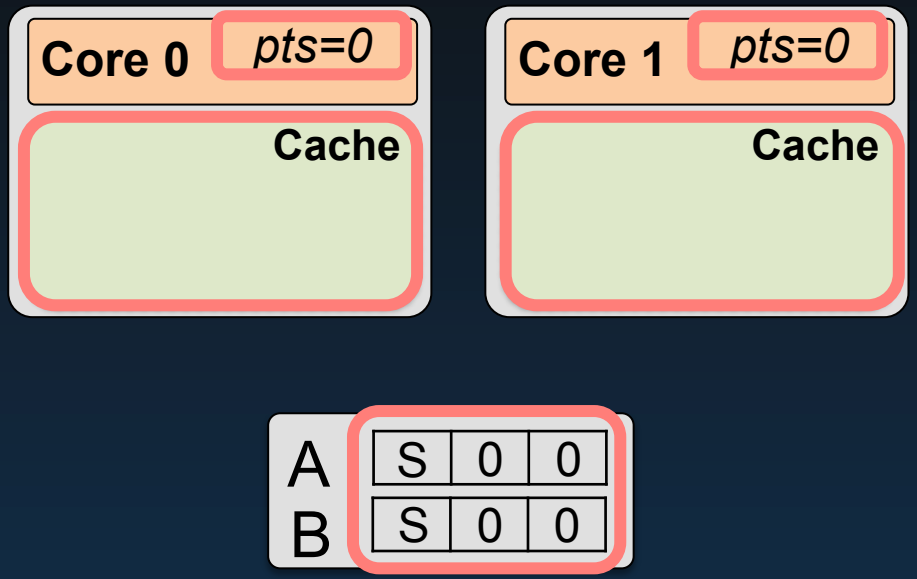
Write Timestamp (*wts*)
Data created at *wts*



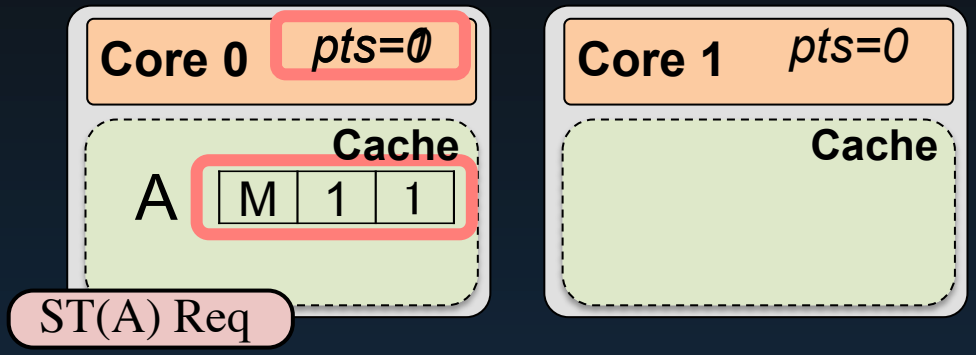
Read Timestamp (*rts*)
Data valid from *wts* to *rts*



Two-Core Example



STORE A @ CORE 0

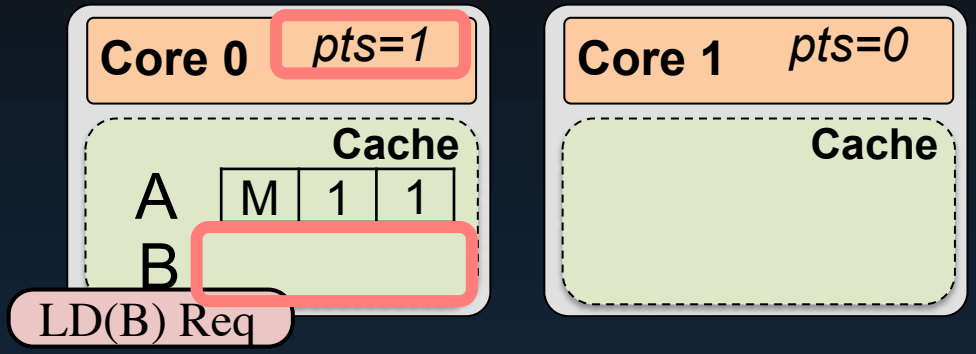


	Core 0	Core 1
1	store A	
2	load B	
3		store B
4		load A
5	load B	

A	M	owner	0
B	S	0	0

Write at *pts = 1*

LOAD B @ CORE 0

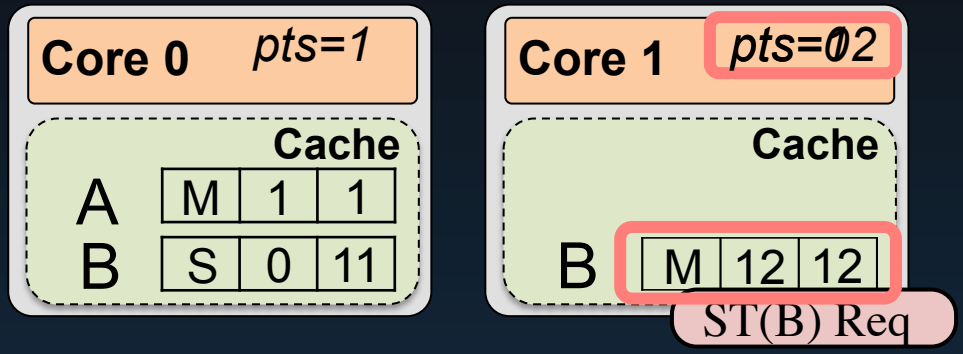


Core 0	Core 1
1 store A	
2 **load B**	
	3 store B
	4 load A
5 load B	

A	M	owner:0	
B	S	0	10 11

Reserve *rts* to *pts* + *lease* = *11*

Store B @ Core 1

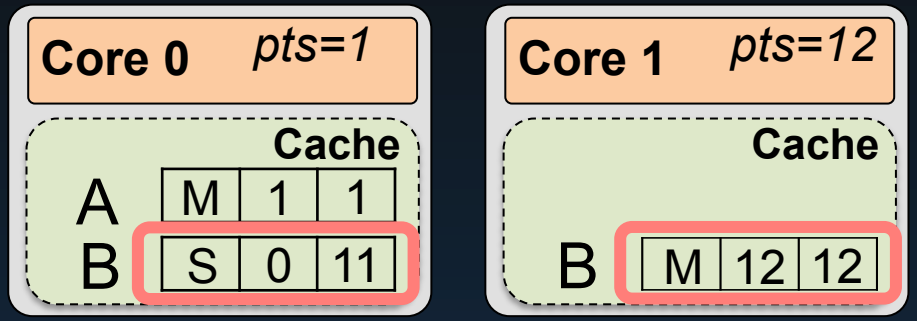


Core 0	Core 1
1 store A	
2 load B	
	3 **store B**
	4 load A
5 load B	

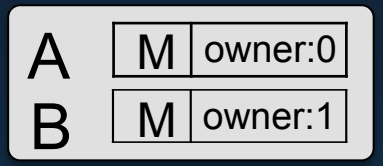
Exclusive ownership returned

No invalidation

Two Versions Coexist



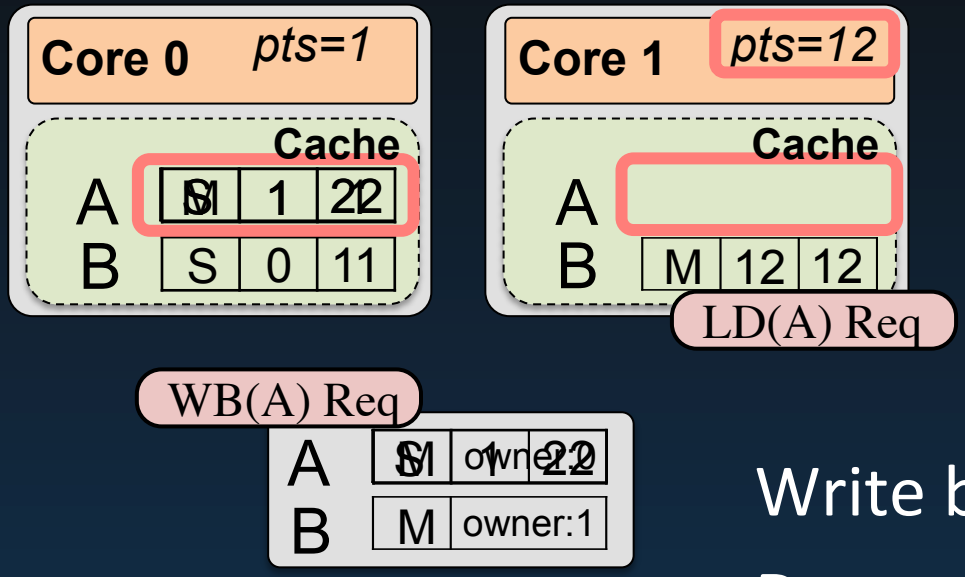
Core 0	Core 1
1 store A	
2 load B	
	3 **store B**
	4 load A
5 load B	



Core 1 traveled ahead in time

Versions ordered in logical time

LOAD A @ CORE 1



Core 0

1. store A
2. load B
5. load B

Core 1

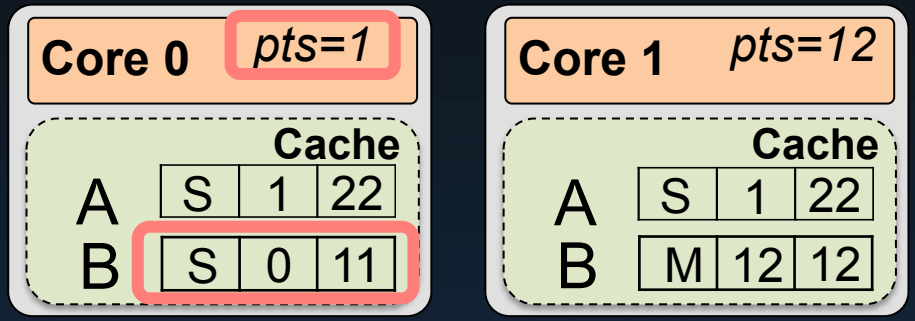
3. store B
4. **load A**

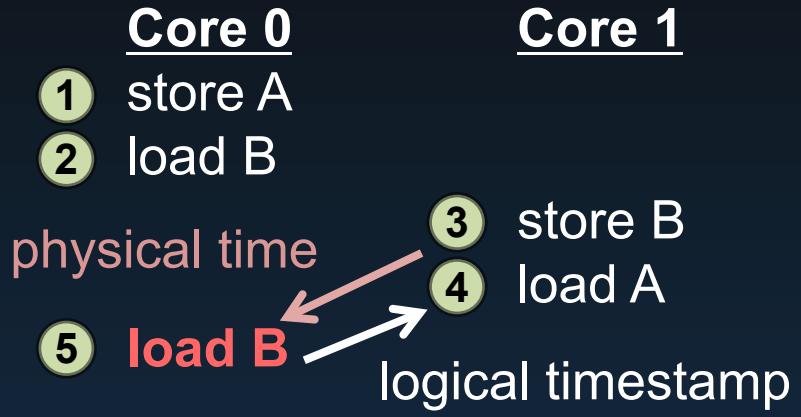
Write back request to Core 0

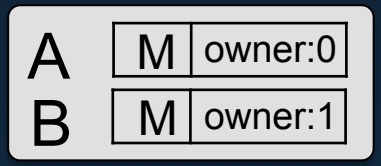
Downgrade from M to S

Reserve *rts* to *pts + lease = 22*

LOAD B @ CORE 0



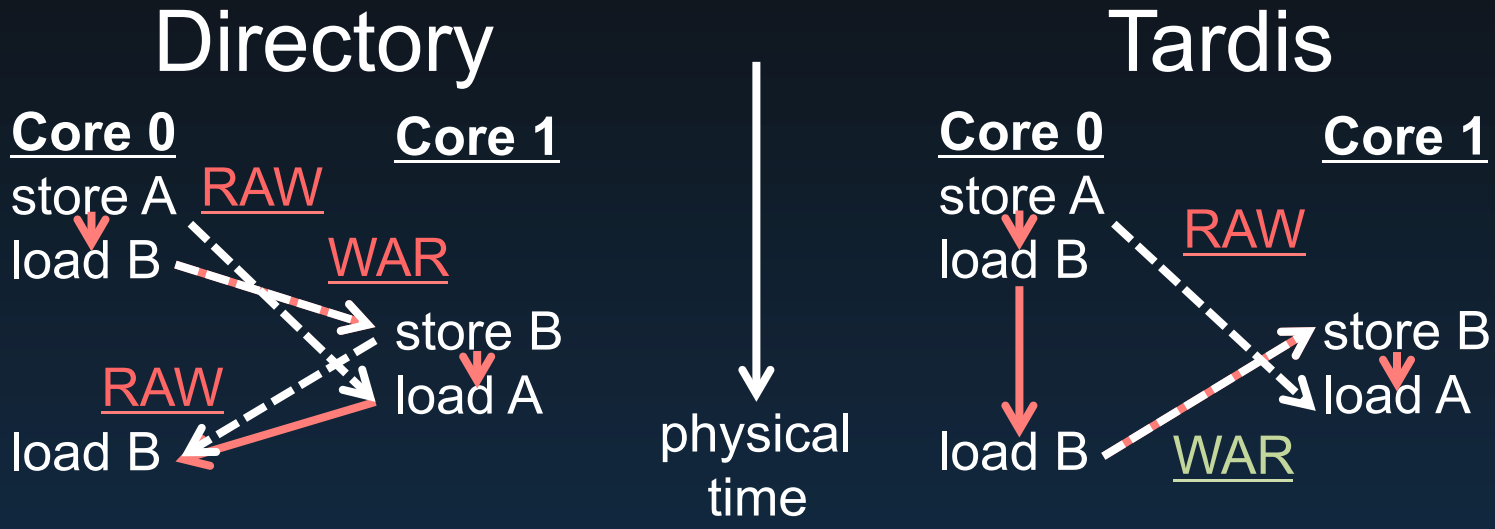




5 > physical time 4, 5 < logical time 4

global memory order ≠ physical time order

Summary of Example

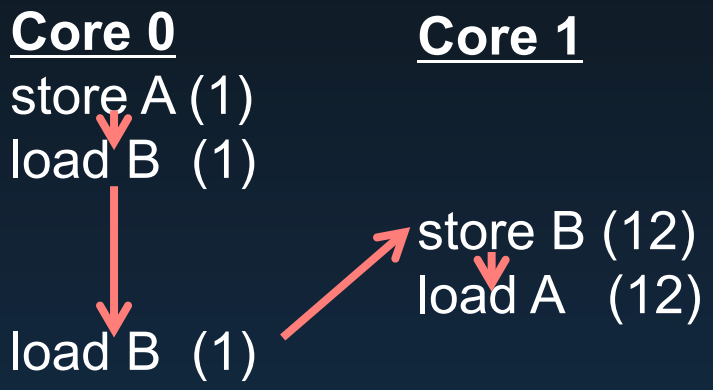


physical time order

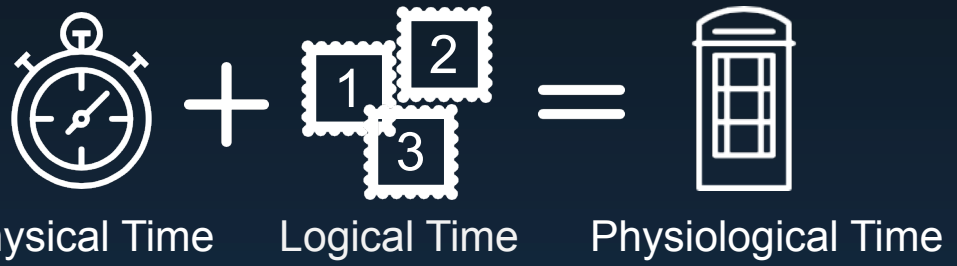
physical + logical time order

Physiological Time

Tardis



Global Memory Order



$$X <_{PL} Y := X <_{L} Y \text{ or } (X =_{L} Y \text{ and } X <_{P} Y)$$

Thm: Tardis obeys Sequential Consistency

Tardis Pros and Cons

Scalability

No Invalidation, Multicast or Broadcast

Lease Renew

Speculative Read

Timestamp Size

Timestamp Compression

Time Stands Still

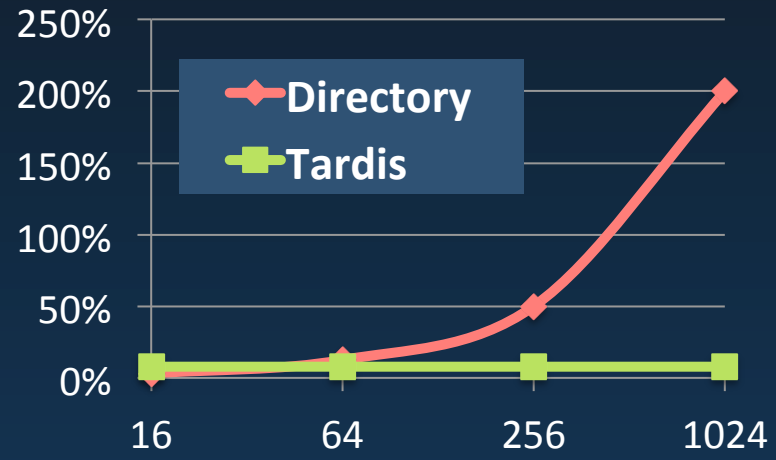
Livelock Avoidance

EVALUATION

Storage overhead per cacheline (N cores)

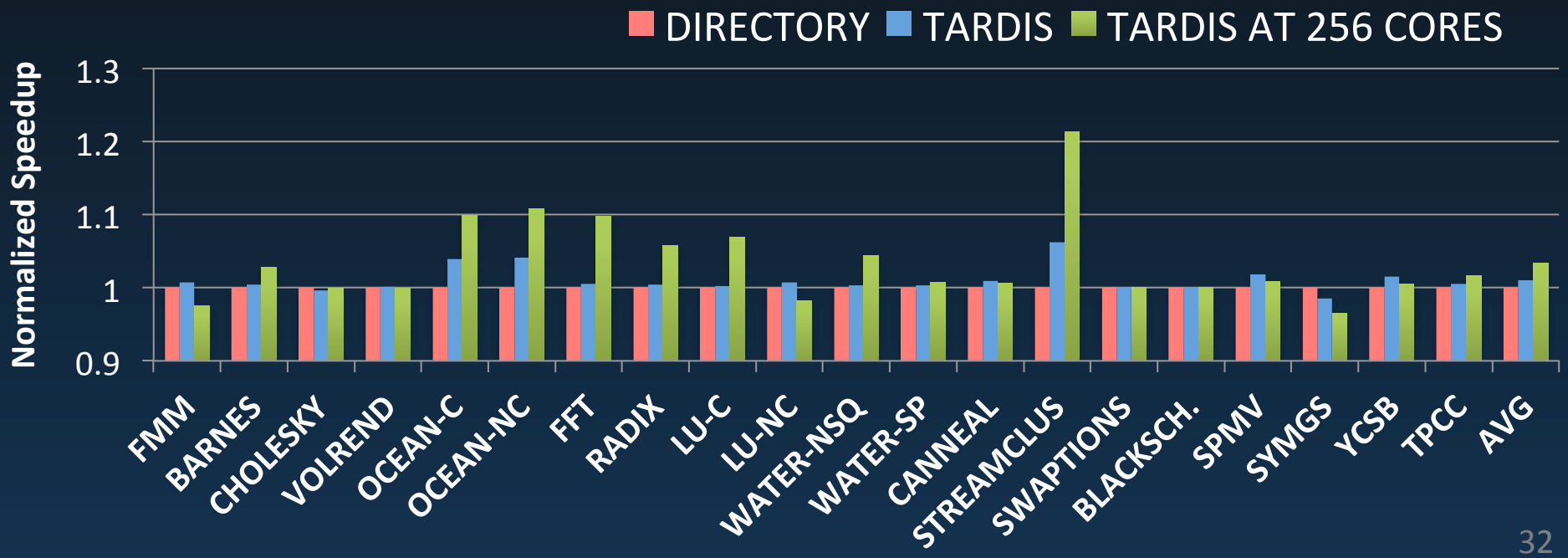
Directory: ***N*** bits per cacheline

Tardis: ***Max(Const, log(N))*** bits per cacheline

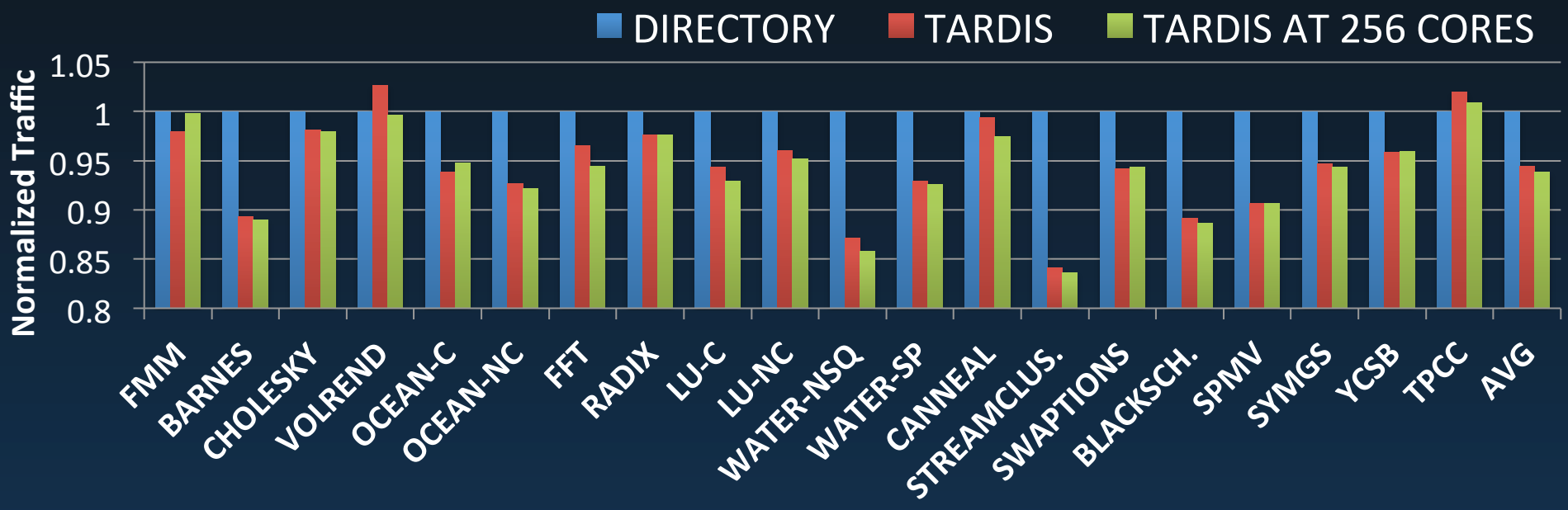


SPEEDUP

Graphite Multi-core Simulator (64 cores)



NETWORK TRAFFIC

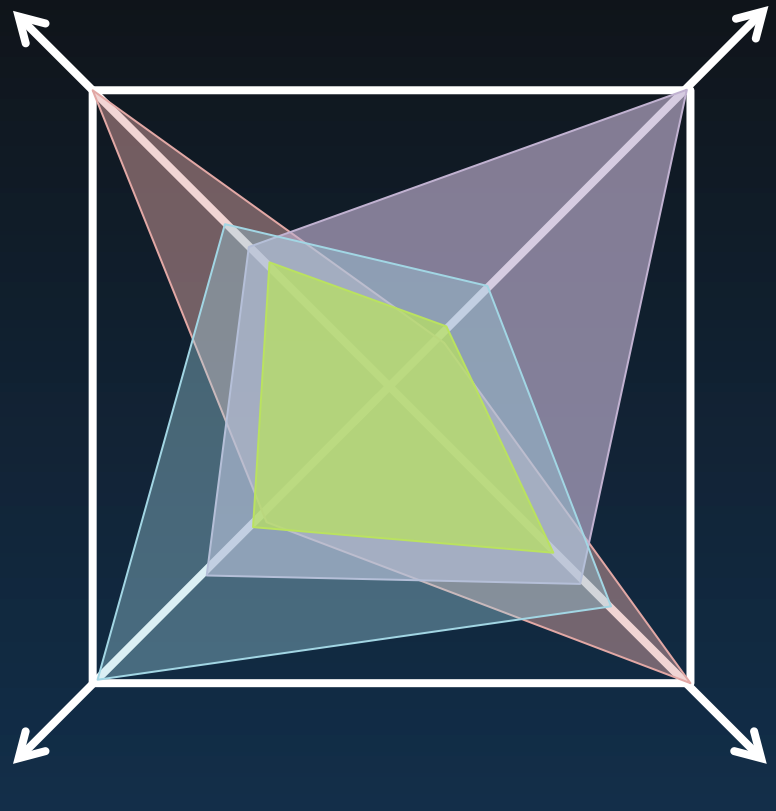


Snoopy Coherence

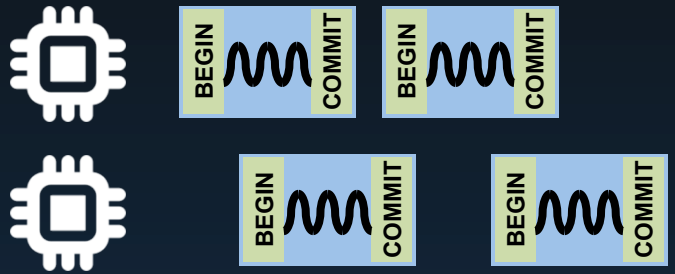
Directory Coherence

Optimized Directory

TARDIS



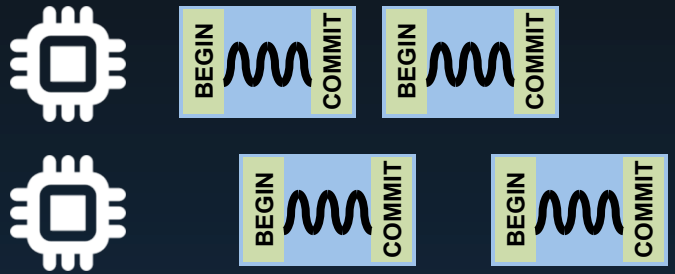
Concurrency Control



Serializability

Results should correspond to some serial order of atomic execution

Concurrency Control

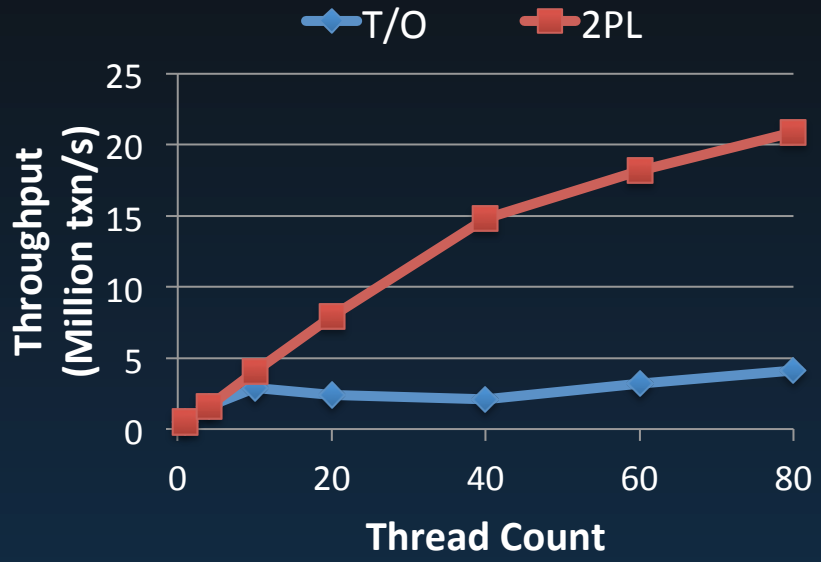


Can't Have This

Results should correspond to some serial order of atomic execution

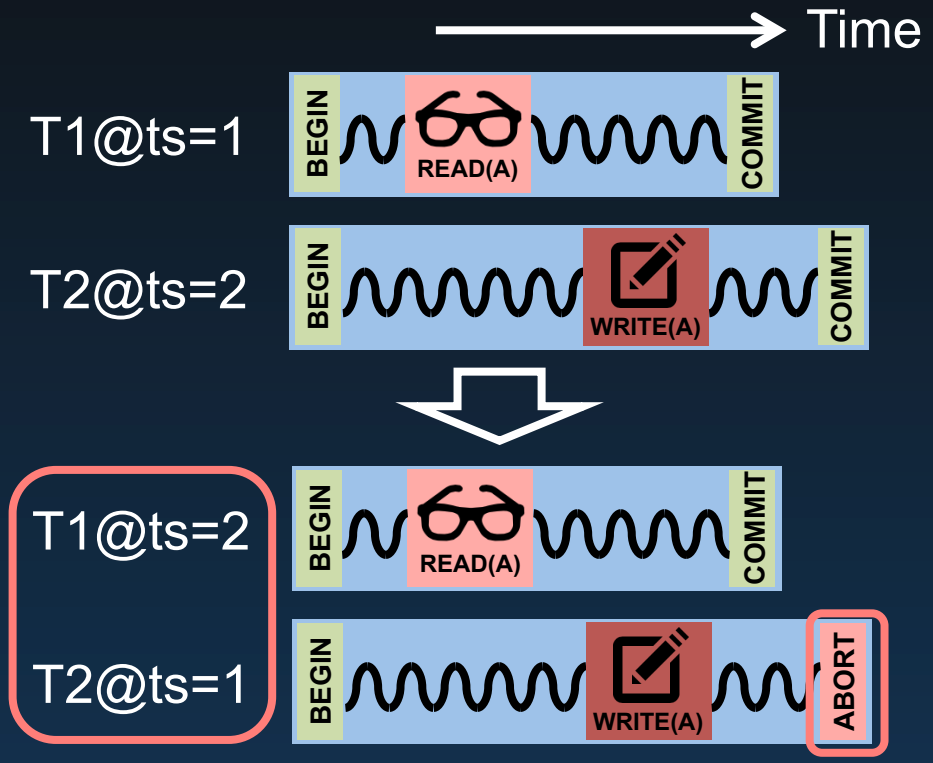
Bottleneck 1: Timestamp Allocation

- Centralized Allocator
  - Timestamp allocation is a scalability bottleneck
- Synchronized Clock
  - Clock skew causes unnecessary aborts



Bottleneck 2: Static Assignment

- Timestamps assigned before a transaction starts
- Suboptimal assignment leads to unnecessary aborts.



Key Idea: Data Driven Timestamp Management

Traditional T/O

1. Acquire timestamp (TS)
2. Determine tuple visibility using TS

- ✗ Timestamp Allocation
- ✗ Static Timestamp Assignment

TicToc

1. Access tuples and remember their timestamp info.
2. Compute commit timestamp (CommitTS)

- ✓ No Timestamp Allocation
- ✓ Dynamic Timestamp Assignment

TicToc Transaction Execution

Read & Write Tuples
Execute Transaction

- Compute CommitTS
- Decide Commit/Abort

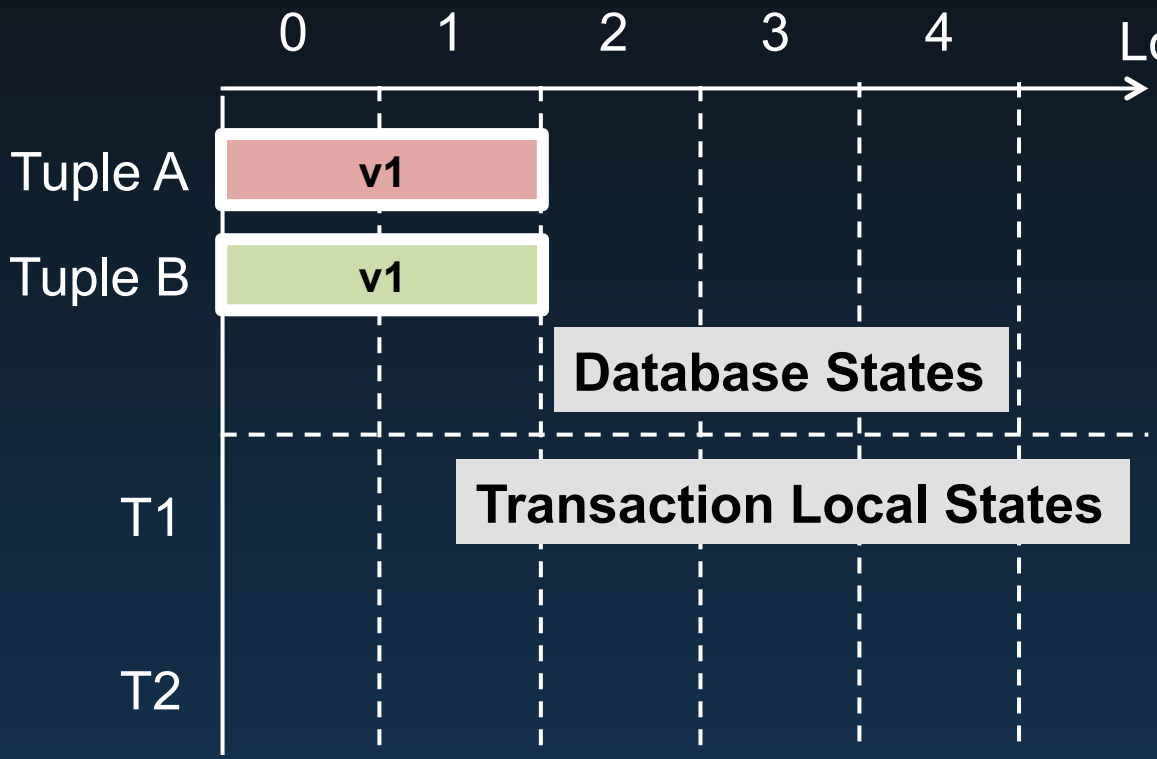
- Update Database

wts : last data write @ wts
rts : last data read @ rts

} data valid between wts and rts

Tuple Format

TicToc Example



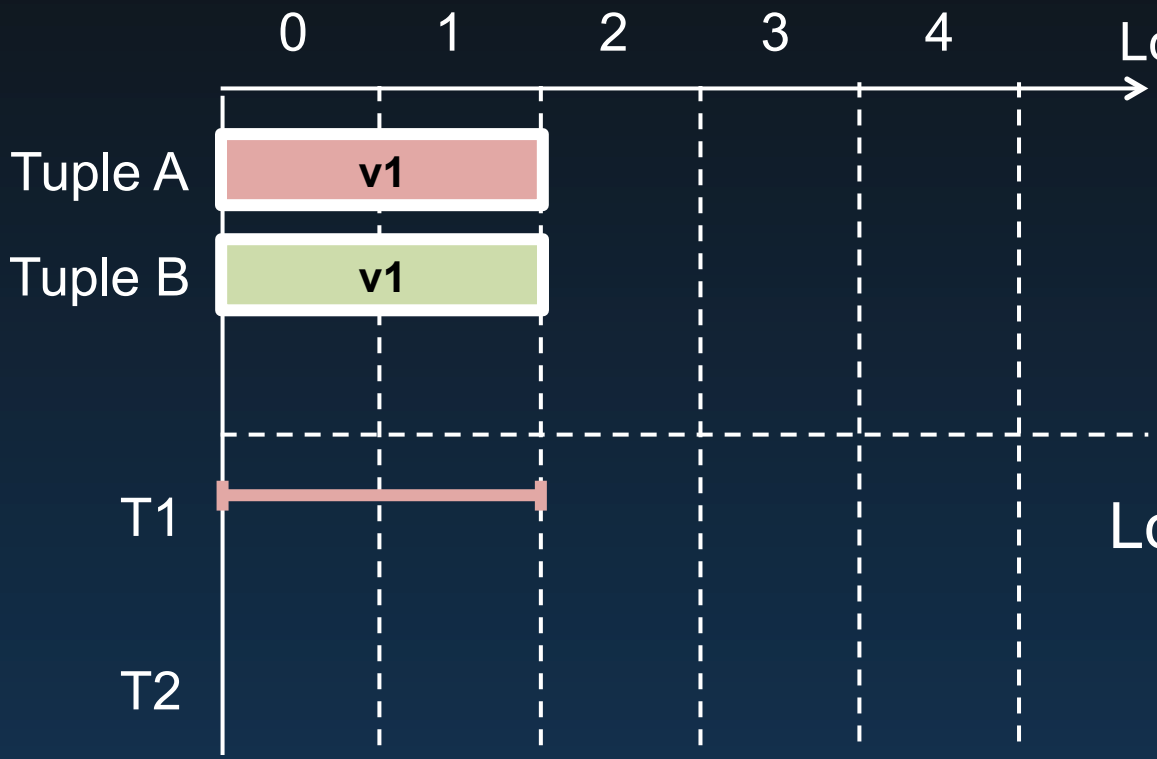
Transaction 1

1. load A
3. store B
5. commit?

Transaction 2

2. load A
4. load B
6. commit?

LOAD A FROM T1



Transaction 1

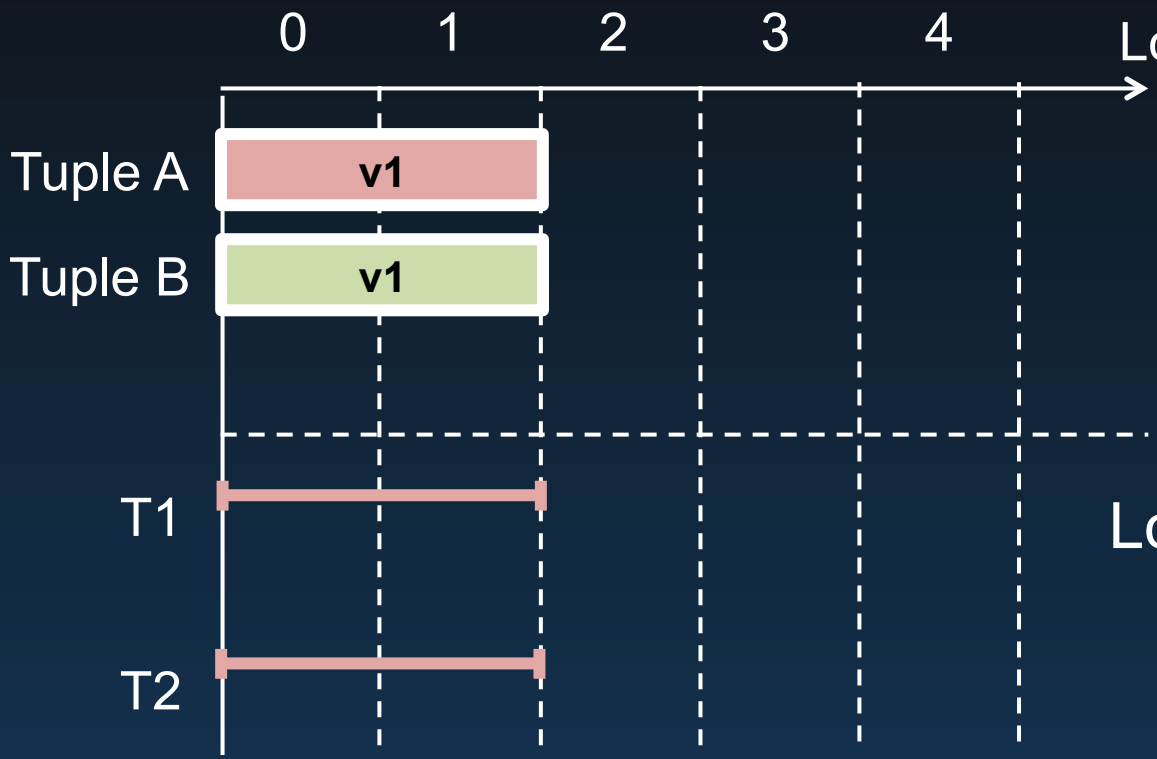
1. load A
3. store B
5. commit?

Transaction 2

2. load A
4. load B
6. commit?

Load a snapshot of tuple A
- Data, wts and rts

LOAD A FROM T2



Transaction 1

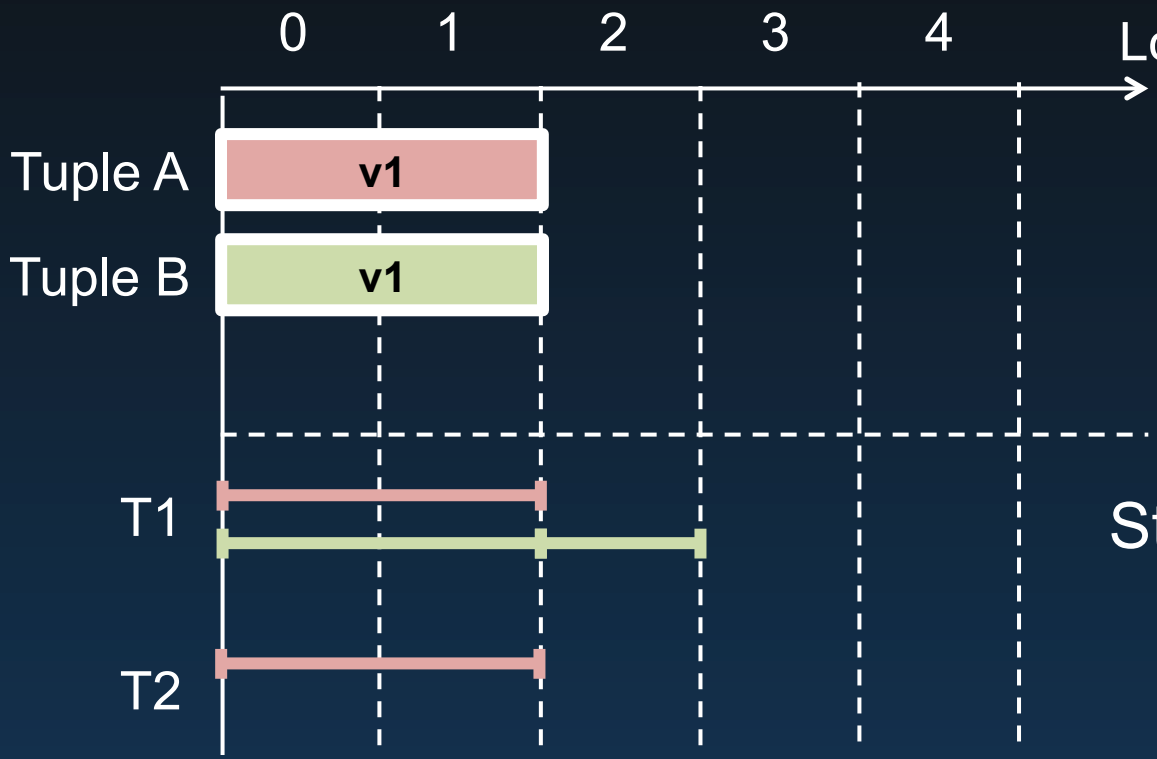
1. load A
3. store B
5. commit?

Transaction 2

2. **load A**
4. load B
6. commit?

Load a snapshot of tuple A
- Data, wts and rts

STORE B FROM T1



Transaction 1

1. load A
3. **store B**
5. commit?

Transaction 2

2. load A
4. load B
6. commit?

Store B to local write set

LOAD B FROM T2

Transaction 1

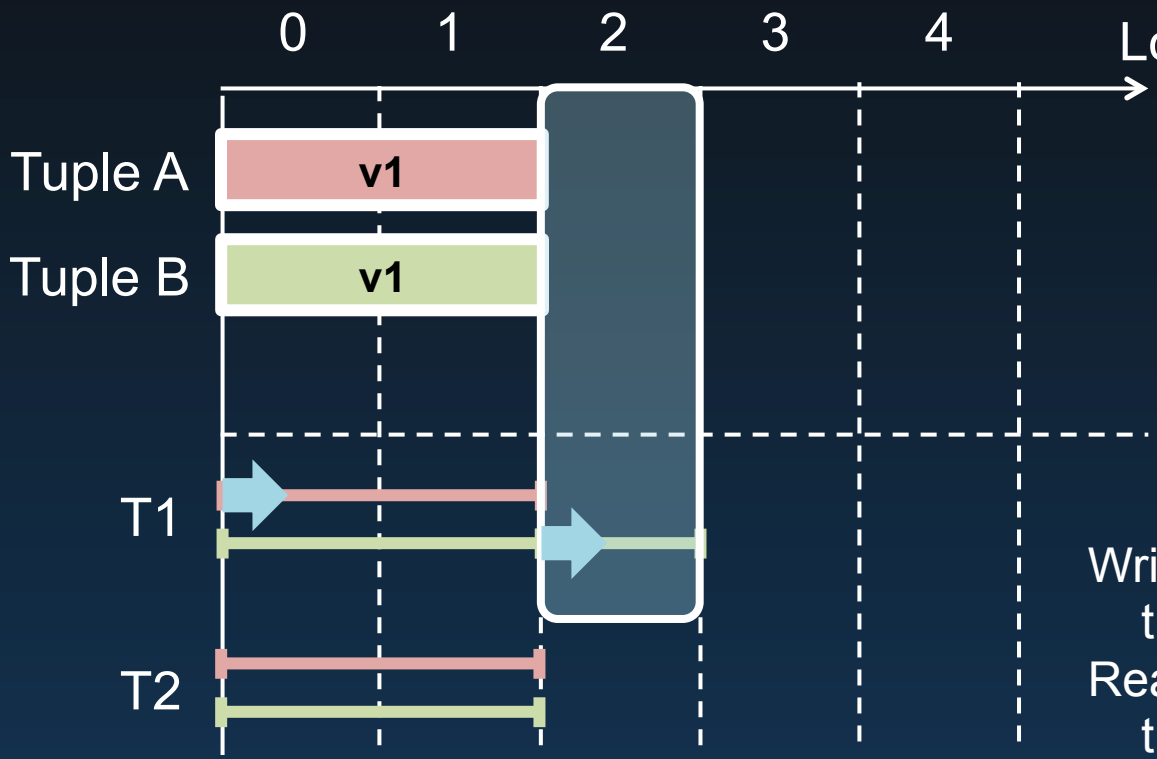
1. load A
3. store B
5. commit?

Transaction 2

2. load A
4. **load B**
6. commit?

Load a snapshot of tuple B
- Data, wts and rts

Commit Phase of T1



Transaction 1	Transaction 2
1 load A	2 load A
3 store B	4 load B
5 commit?	6 commit?

Compute CommitTS

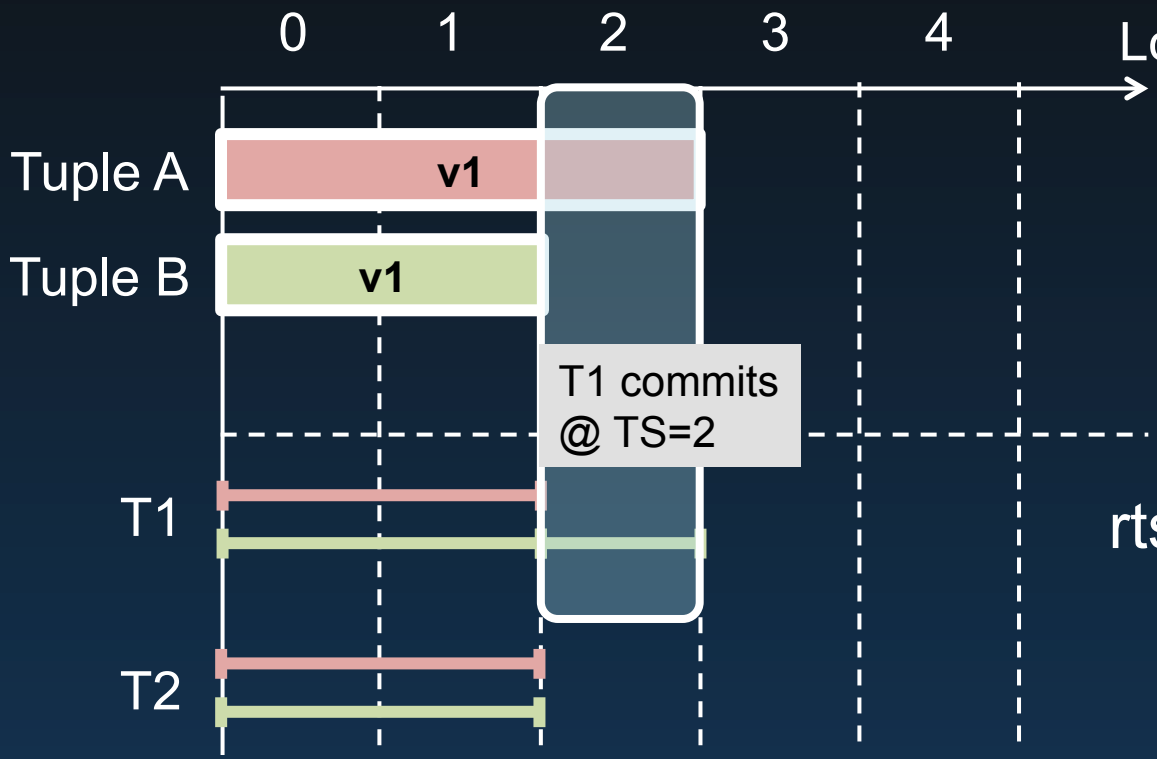
Write Set:

tuple.rts+1 ≤ CommitTs

Read Set:

tuple.wts ≤ CommitTs ≤ tuple.rts

COMMIT PHASE OF T1



Transaction 1

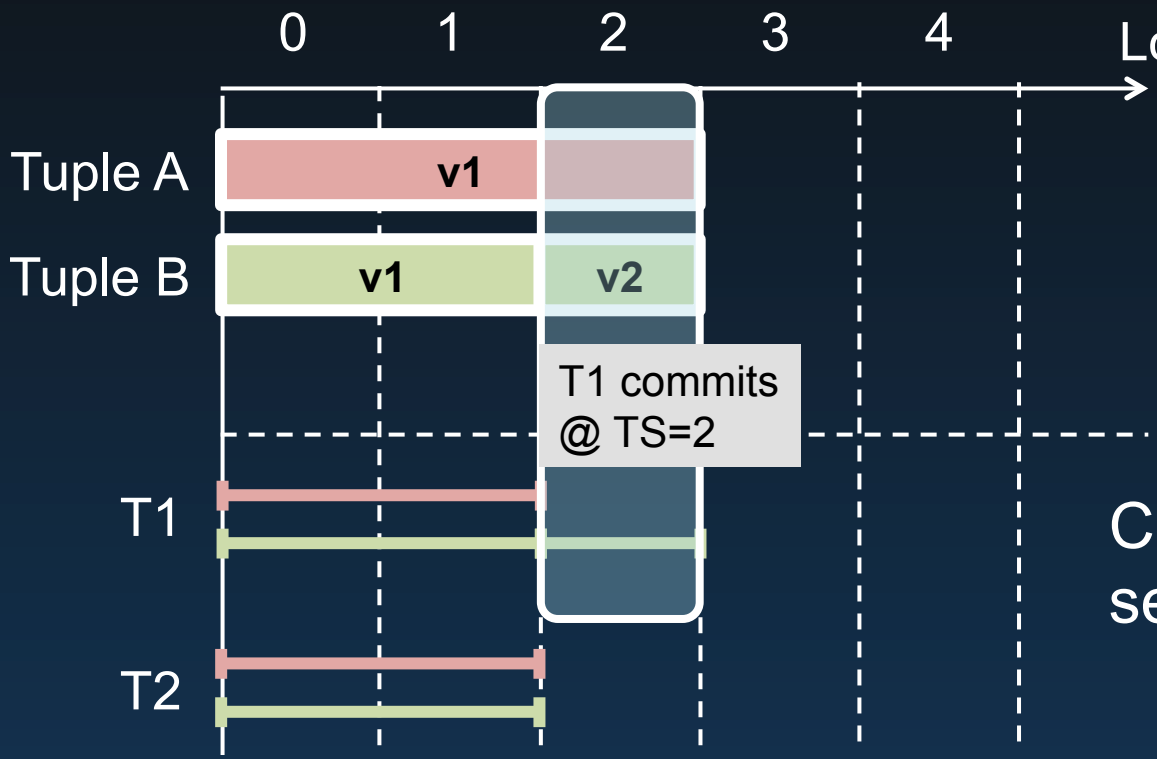
1. load A
3. store B
5. commit?

Transaction 2

2. load A
4. load B
6. commit?

rts extension on tuple A

Commit Phase of T1



Transaction 1

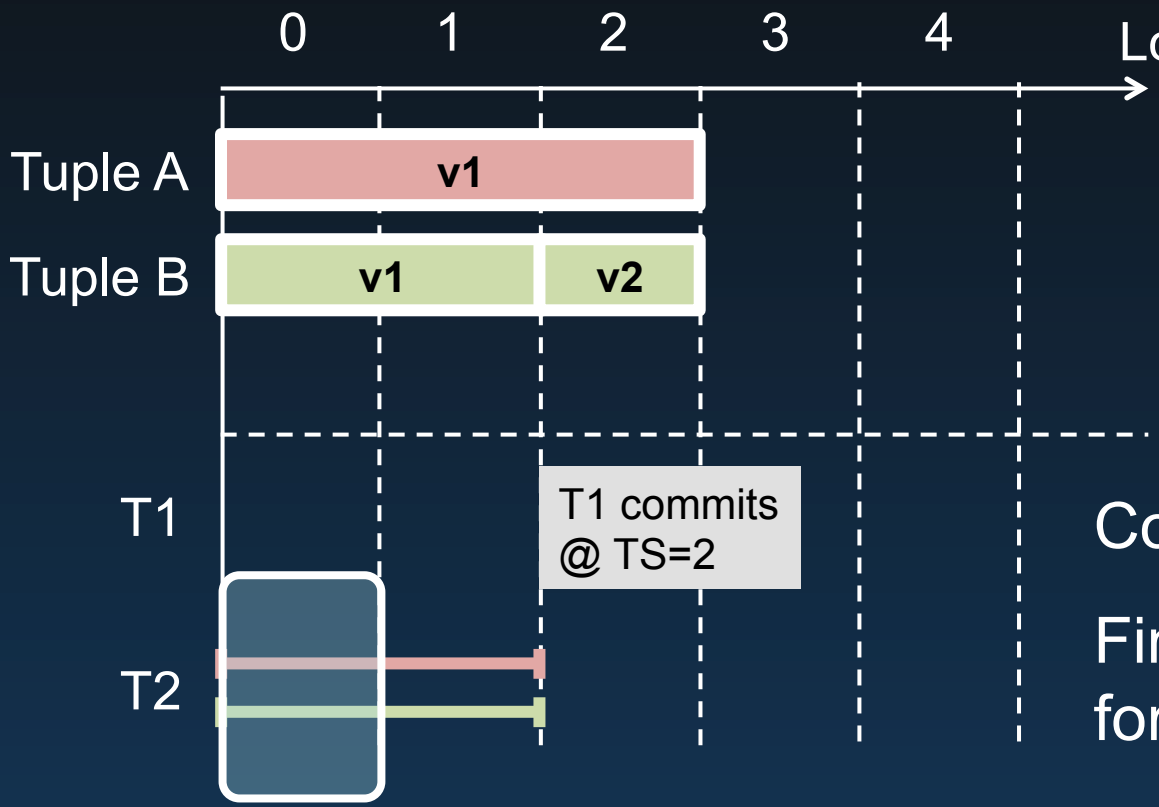
1. load A
3. store B
5. commit?

Transaction 2

2. load A
4. load B
6. commit?

Copy tuple B from write set to database

COMMIT PHASE OF T2



Transaction 1

1. load A
3. store B
5. commit?

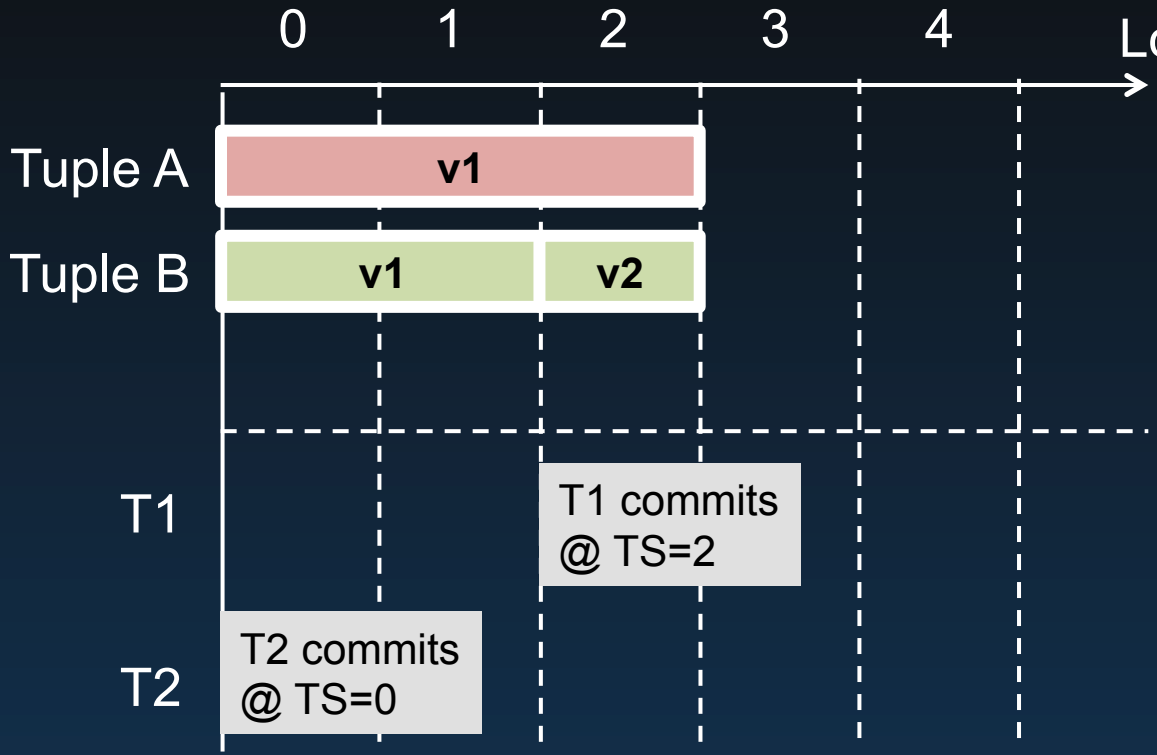
Transaction 2

2. load A
4. load B
6. **commit?**

Compute CommitTS for T2

Find consistent read time for T2 (no writes in T2)

FINAL STATE



Transaction 1

1. load A
3. store B
5. commit?

Transaction 2

2. load A
4. load B
6. commit?

Txn 1 $<_{\text{physical time}}$ Txn 2

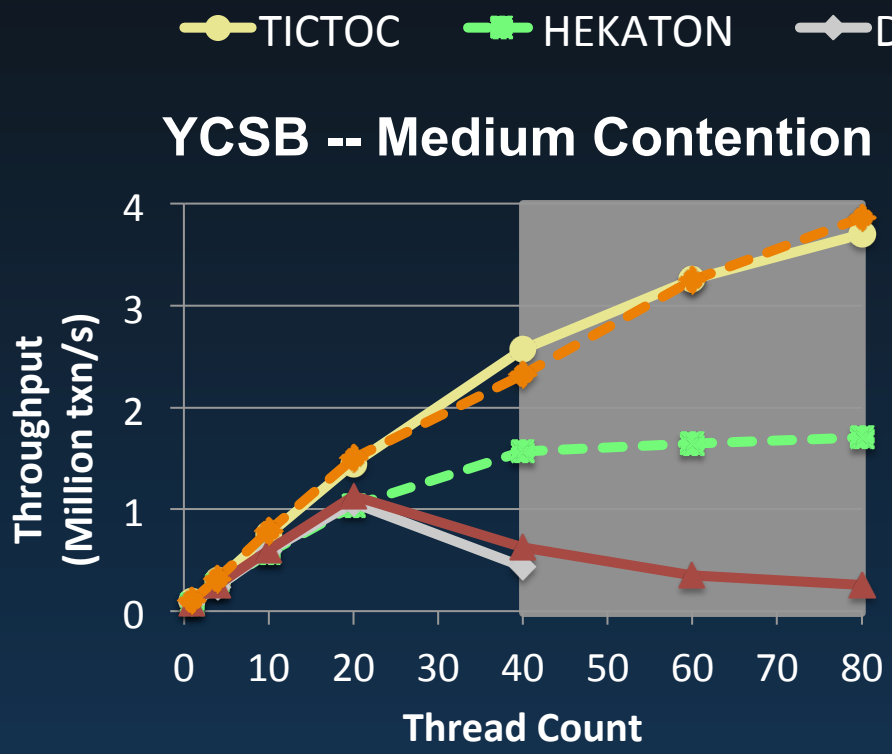
Txn 2 $<_{\text{logical time}}$ Txn 1

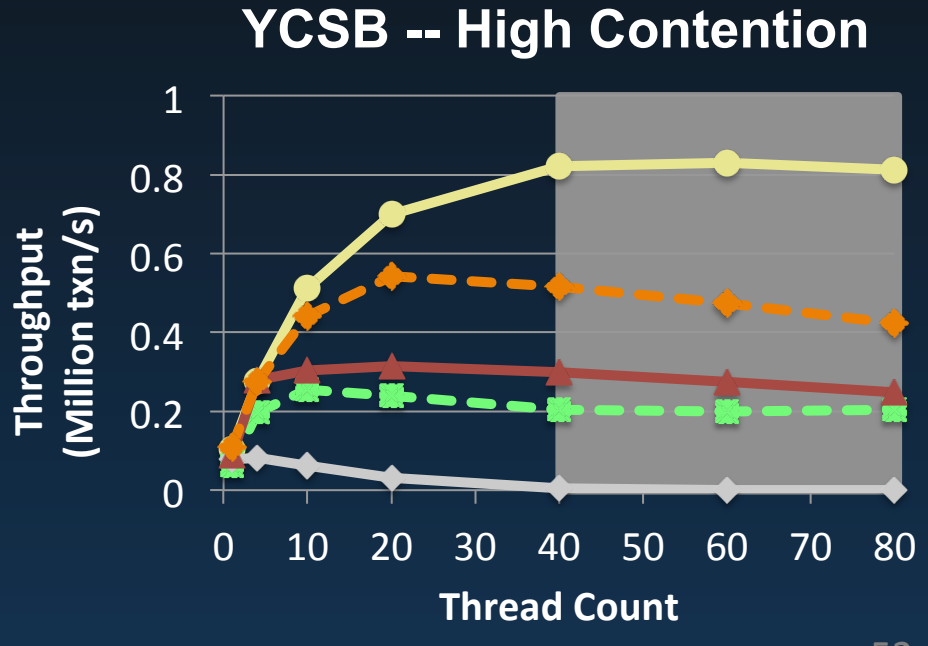
Thm: Serializability = All operations valid at CommitTS

Experimental Setup

- DBx1000: Main Memory DBMS
  - No logging
  - No B-tree (hash indexing)
- Concurrency Control Algorithms
  - MVCC: HEKATON (Microsoft)
  - OCC: SILO (Harvard/MIT)
  - 2PL: DL_DETECT, NO_WAIT
- 10 GB YCSB Benchmark

EVALUATION


TICTOC
HEKATON
DL_DETECT
NO_WAIT
SILO
YCSB -- Medium Contention
Throughput (Million txn/s)
0
1
2
3
4
0
10
20
30
40
50
60
70
80
Thread Count

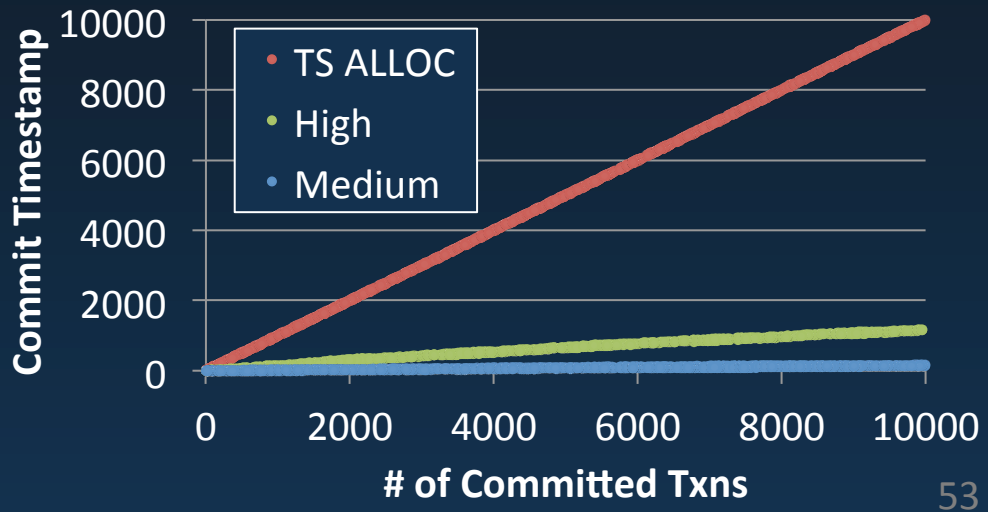

YCSB -- High Contention
Throughput (Million txn/s)
0
0.2
0.4
0.6
0.8
1
0
10
20
30
40
50
60
70
80
Thread Count

TicToc Discussion

Thm: Serializability = All ops valid at CommitTS

Transactions may have same CommitTS

Logical timestamp growing rate indicates inherent parallelism



Physiological Time Across the Stack

Circuit — Efficient atomic instructions

Multicore Processor — Tardis coherence protocol

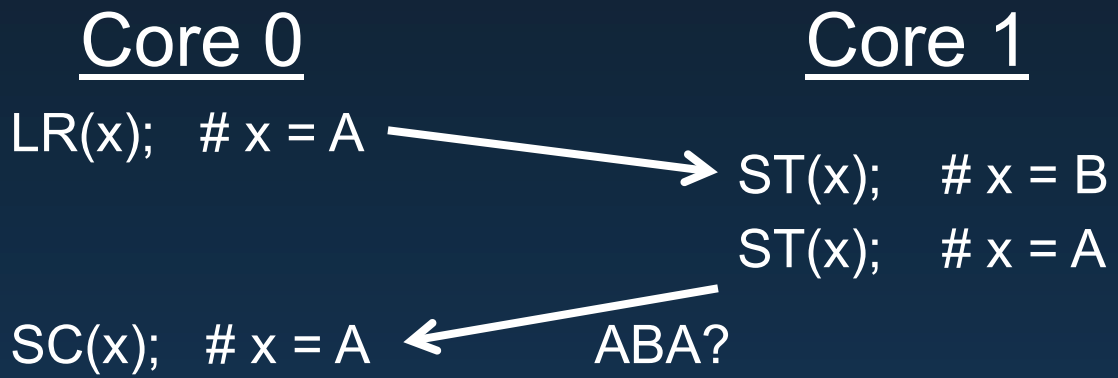
Multicore Database — TicToc concurrency control

Distributed Database — Distributed TicToc

Distributed Shared Memory — Transaction processing with fault tolerance

Atomic Instruction (LR/SC)

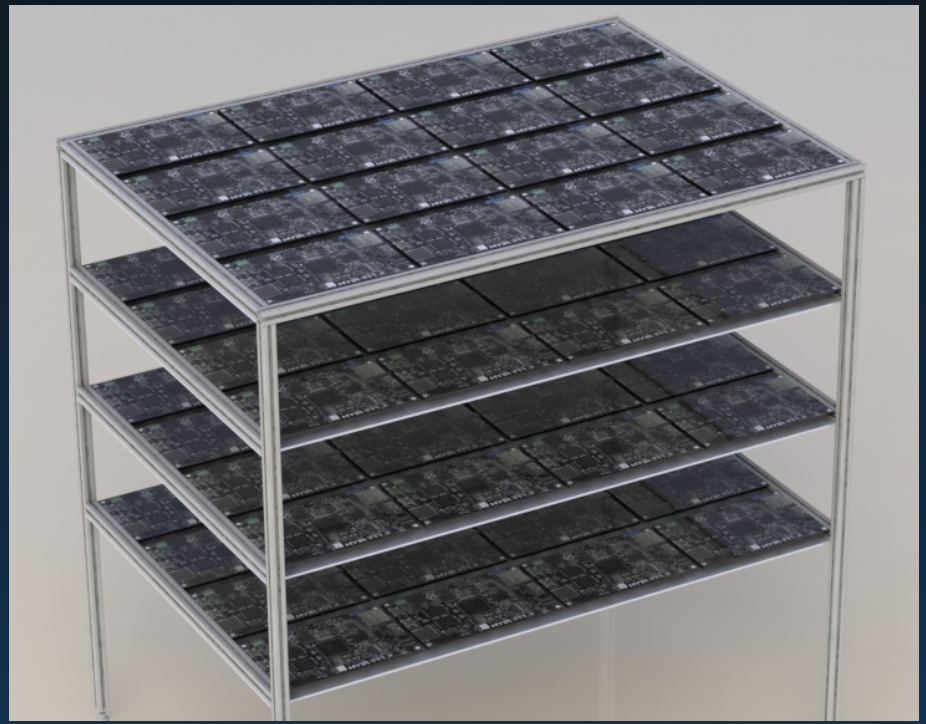
- ABA Problem
- Detect ABA using timestamp (wts)



TARDIS CACHE COHERENCE

- Simple: No Invalidation
- Scalable:
  - *O(log N)* storage
  - No Broadcast, No Multicast
  - No Clock Synchronization
- Support Relaxed Consistency Models

T1000: Proposed 1000-Core Shared Memory Processor



TicToc Concurrency Control

- Data Driven Timestamp Management
- No Central Timestamp Allocation
- Dynamic Timestamp Assignment

Distributed TicToc

- Data Driven Timestamp Management
- Efficient Two-Phase Commit Protocol
- Support Local Caching of Remote Data

Fault Tolerant Distributed Shared Memory

- Transactional Programming Model
- Distributed Command Logging
- Dynamic Dependency Tracking Among Transactions (WAR dependency can be ignored)

TIME TRAVELING TO ELIMINATE WAR

Xiangyao Yu, Srini Devadas
CSAIL, MIT