CS 6530: Advanced Database Systems Fall 2022

Lecture 07
Concurrency control #2

Prashant Pandey

prashant.pandey@utah.edu

Acknowledgement: Slides taken from Prof. Andy Pavlo, CMU

SCHOOL OF COMPUTING
UNIVERSITY OF UTAH

CONCURRENCY CONTROL

- The system assumes that a txn could stall at any time whenever it tries to access data that is not in memory.

- Execute other txns at the same time so that if one txn stalls then others can keep running.
  - Set locks to provide ACID guarantees for txns.
  - Locks are stored in a separate data structure to avoid being swapped to disk.

Acknowledgement: Prof. Andy Pavlo, CMU

ACID guarantee

- **Atomicity** - each statement in a transaction (to read, write, update or delete data) is treated as a single unit. Either the entire statement is executed, or none of it is executed.
- **Consistency** - ensures that transactions only make changes to tables in predefined, predictable ways
- **Isolation** - when multiple users are reading and writing from the same table all at once, isolation of their transactions ensures that the concurrent transactions don’t interfere with or affect one another.
- **Durability** - ensures that changes to your data made by successfully executed transactions will be saved, even in the event of system failure.

STORAGE ACCESS LATENCIES

	L3	*DRAM*	*SSD*	*HDD*
Read Latency	~20 ns	60 ns	25,000 ns	10,000,000 ns
Write Latency	~20 ns	60 ns	300,000 ns	10,000,000 ns

Acknowledgement: Prof. Andy Pavlo, CMU

CONCURRENCY CONTROL

- The protocol to allow txns to access a database in a multi-programmed fashion while preserving the illusion that each of them is executing alone on a dedicated system.
  - The goal is to have the effect of a group of txns on the database's state is equivalent to any serial execution of all txns.

- Provides **A**tomicity + **I**solation in ACID

Acknowledgement: Prof. Andy Pavlo, CMU

CONCURRENCY CONTROL

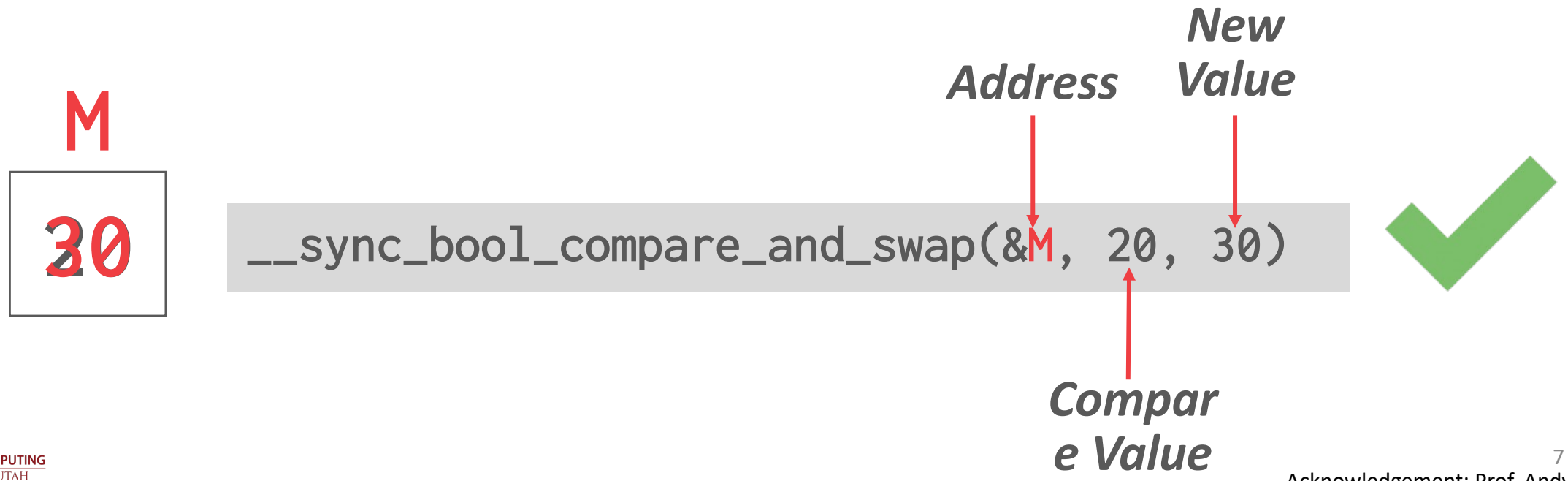
- For in-memory DBMSs, the cost of a txn acquiring a lock is the same as accessing data.
- New bottleneck is contention caused from txns trying access data at the same time.

- The DBMS can store locking information about each tuple together with its data.
  - This helps with CPU cache locality.
  - Mutexes are too slow. Need to use **compare-and-swap** (CAS) instructions.

Acknowledgement: Prof. Andy Pavlo, CMU

COMPARE-AND-SWAP

- Atomic instruction that compares contents of a memory location M to a given value V
  - If values are equal, installs new given value V' in M
  - Otherwise operation fails



Acknowledgement: Prof. Andy Pavlo, CMU

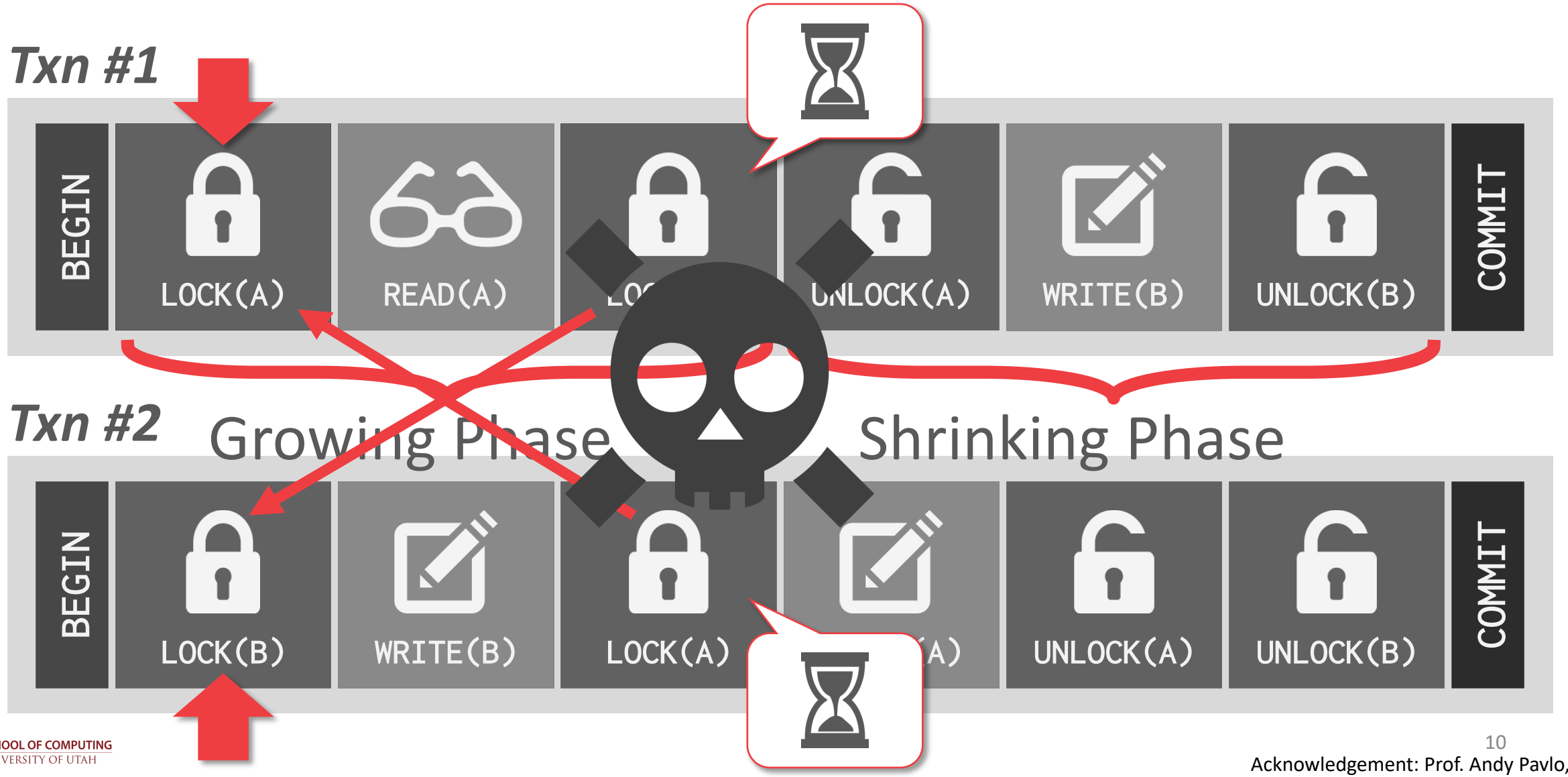
CONCURRENCY CONTROL SCHEMES

- **Two-Phase Locking (2PL)**
  - Assume txns will conflict so they must acquire locks on database objects before they are allowed to access them.

- **Timestamp Ordering (T/O)**
  - Assume that conflicts are rare so txns do not need to first acquire locks on database objects and instead check for conflicts at commit time.

Acknowledgement: Prof. Andy Pavlo, CMU

TWO-PHASE LOCKING



Acknowledgement: Prof. Andy Pavlo, CMU

TWO-PHASE LOCKING

- **Deadlock Detection**
  - Each txn maintains a queue of the txns that hold the locks that it waiting for.
  - A separate thread checks these queues for deadlocks.
  - If deadlock found, use a heuristic to decide what txn to kill in order to break deadlock.
- **Deadlock Prevention**
  - Check whether another txn already holds a lock when another txn requests it.
  - If lock is not available, the txn will either (1) wait, (2) commit suicide, or (3) kill the other txn.

Acknowledgement: Prof. Andy Pavlo, CMU

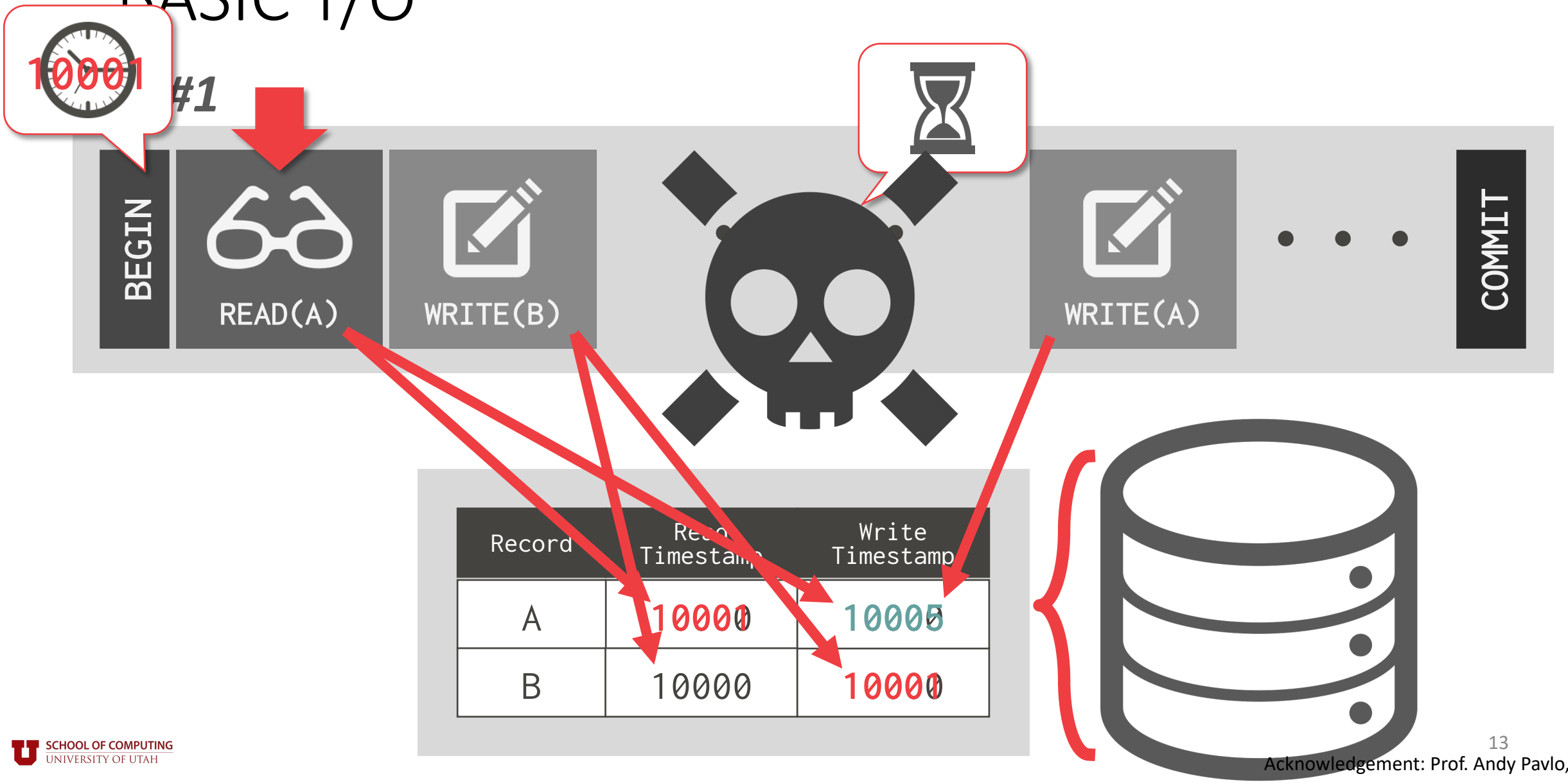
TIMESTAMP ORDERING

- **Basic T/O**
  - Check for conflicts on each read/write.
  - Copy tuples on each access to ensure repeatable reads.

- **Optimistic Currency Control (OCC)**
  - Store all changes in private workspace.
  - Check for conflicts at commit time and then merge.

Acknowledgement: Prof. Andy Pavlo, CMU

BASIC T/O



Acknowledgement: Prof. Andy Pavlo, CMU

OPTIMISTIC CONCURRENCY CONTROL

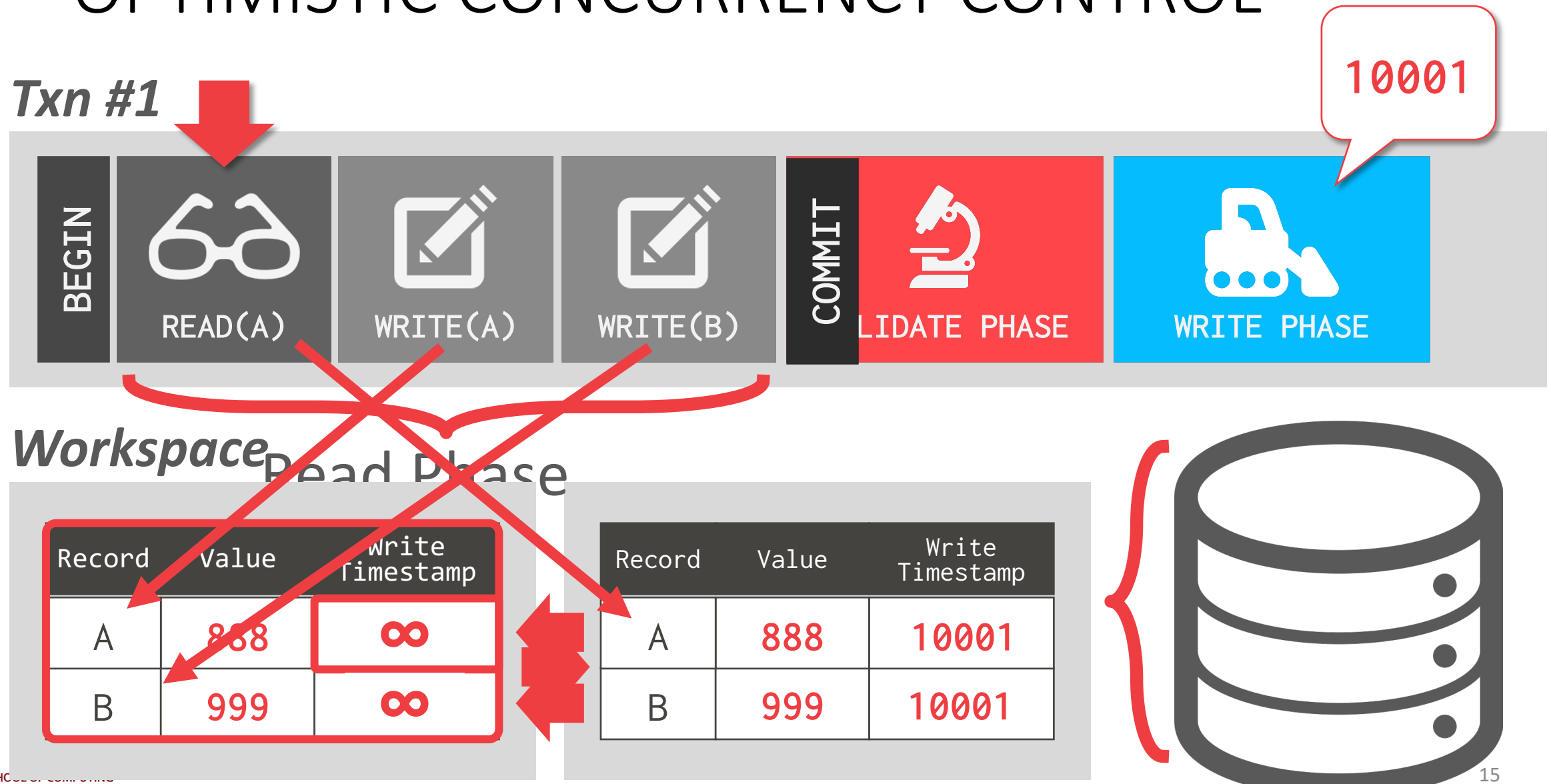
- Timestamp-ordering scheme where txns copy data read/write into a private workspace that is not visible to other active txns.
- When a txn commits, the DBMS verifies that there are no conflicts.

- First proposed in 1981 at CMU by H.T. Kung.

ON OPTIMISTIC METHODS FOR CONCURRENCY CONTROL
ACM TRANSACTIONS ON DATABASE SYSTEMS 1981

Acknowledgement: Prof. Andy Pavlo, CMU

OPTIMISTIC CONCURRENCY CONTROL



Acknowledgement: Prof. Andy Pavlo, CMU

OBSERVATION

- When there is low contention, optimistic protocols perform better because the DBMS spends less time checking for conflicts.

- At high contention, the both classes of protocols degenerate to essentially the same serial execution.

Acknowledgement: Prof. Andy Pavlo, CMU

CONCURRENCY CONTROL EVALUATION

- Compare in-memory concurrency control protocols at high levels of parallelism.
  - Single test-bed system.
  - Evaluate protocols using core counts beyond what is available on today's CPUs.

- Running in extreme environments exposes what are the main bottlenecks in the DBMS.

Acknowledgement: Prof. Andy Pavlo, CMU

1000-CORE CPU SIMULATOR

- DBx1000 Database System
  - In-memory DBMS with pluggable lock manager.
  - No network access, logging, or concurrent indexes.
  - All txns execute using stored procedures.

- MIT Graphite CPU Simulator
  - Single-socket, tile-based CPU.
  - Shared L2 cache for groups of cores.
  - Tiles communicate over 2D-mesh network.

Acknowledgement: Prof. Andy Pavlo, CMU

TARGET WORKLOAD

- Yahoo! Cloud Serving Benchmark (YCSB)
  - 20 million tuples
  - Each tuple is 1KB (total database is ~20GB)
- Each transactions reads/modifies 16 tuples.
- Varying skew in transaction access patterns.
- Serializable isolation level.

Acknowledgement: Prof. Andy Pavlo, CMU

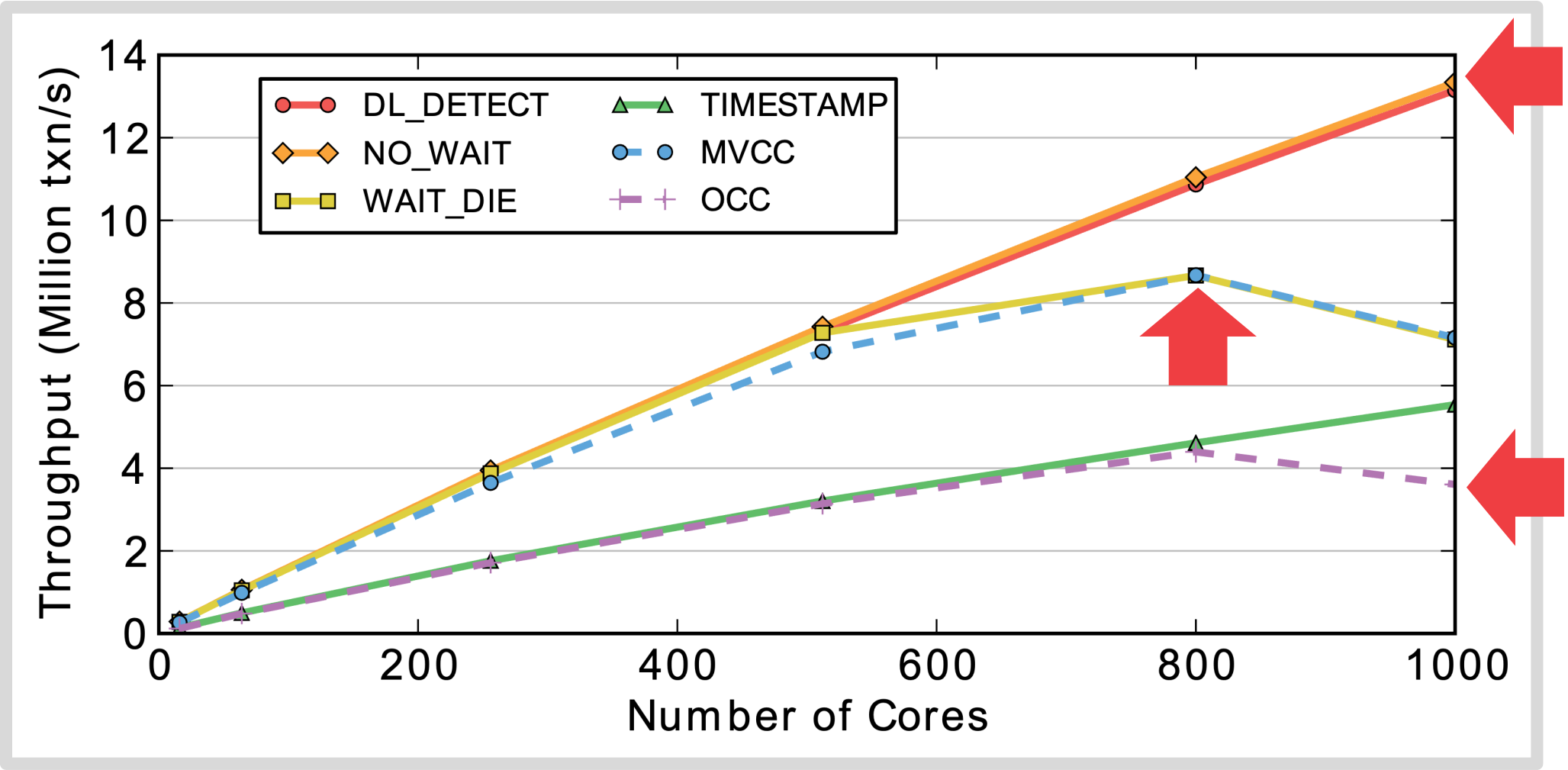
CONCURRENCY CONTROL SCHEMES

DL_DETECT	2PL w/ Deadlock Detection
NO_WAIT	2PL w/ Non-waiting Prevention
WAIT_DIE	2PL w/ Wait-and-Die Prevention

TIMESTAMP	Basic T/O Algorithm
MVCC	Multi-Version T/O
OCC	Optimistic Concurrency Control

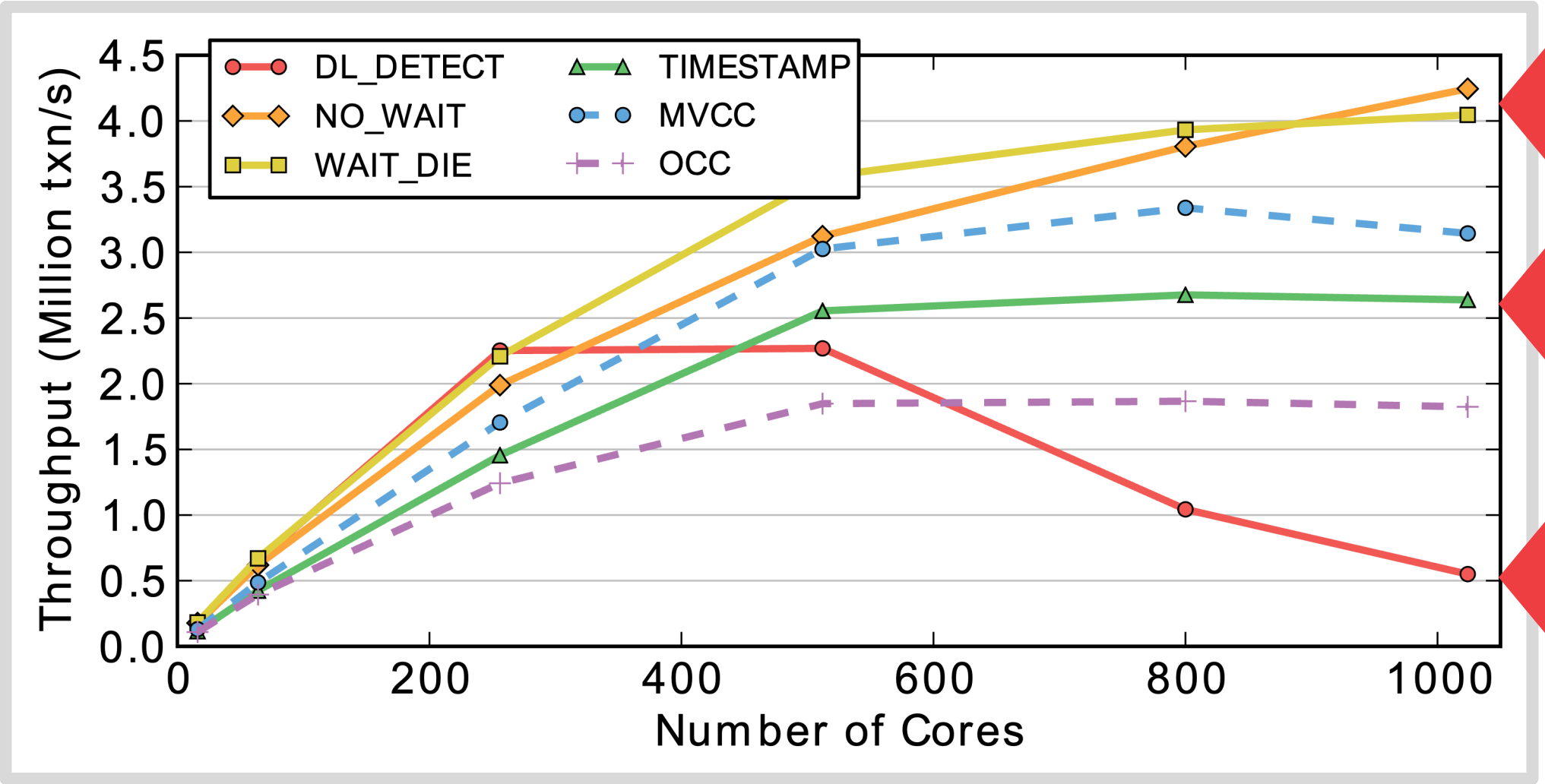
Acknowledgement: Prof. Andy Pavlo, CMU

READ-ONLY WORKLOAD



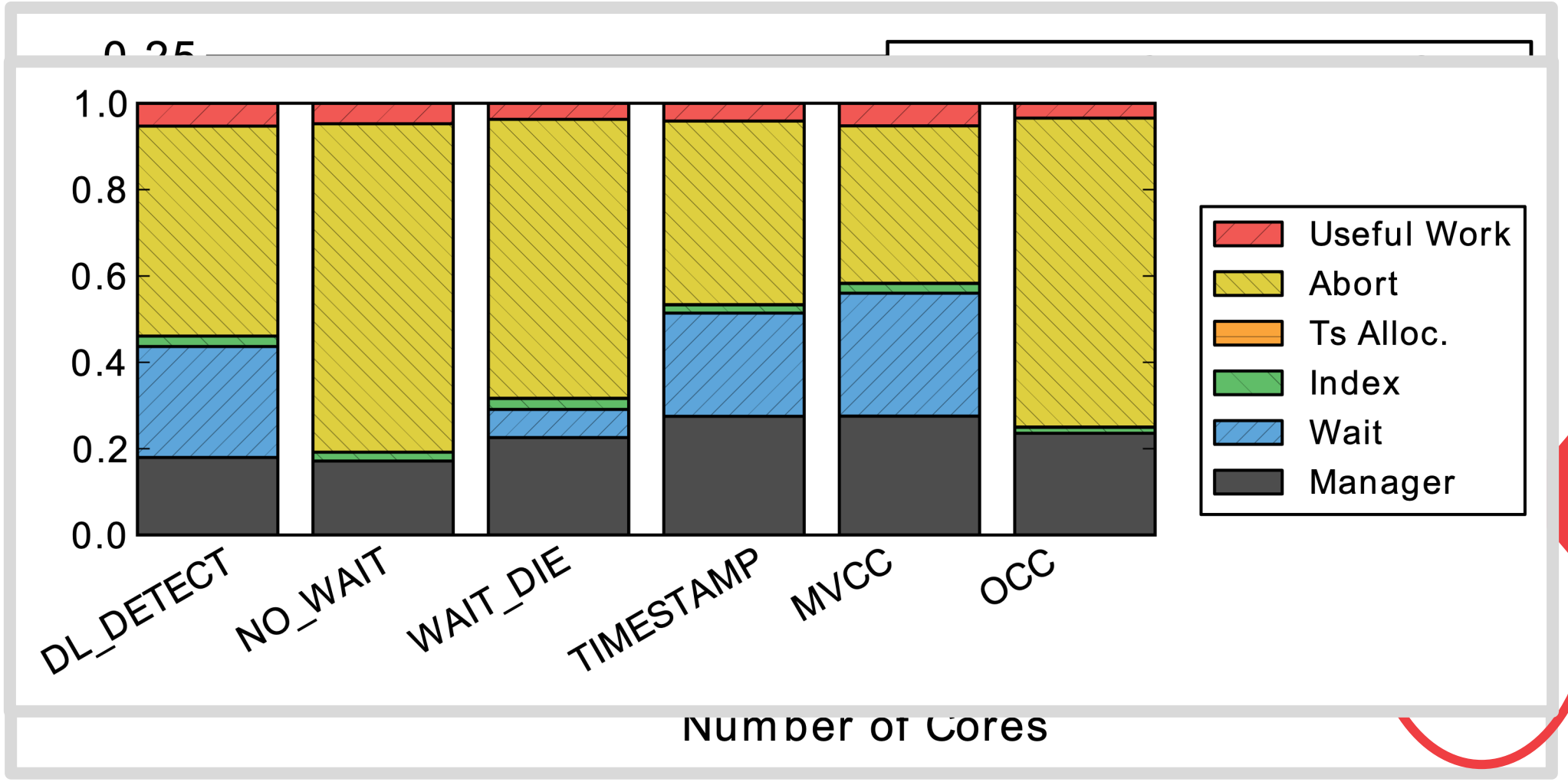
Acknowledgement: Prof. Andy Pavlo, CMU

WRITE-INTENSIVE / MEDIUM-CONTENTION



Acknowledgement: Prof. Andy Pavlo, CMU

WRITE-INTENSIVE / HIGH-CONTENTION



Acknowledgement: Prof. Andy Pavlo, CMU

BOTTLENECKS

- **Lock Thrashing**
  - DL_DETECT, WAIT_DIE
- **Timestamp Allocation**
  - All T/O algorithms + WAIT_DIE
- **Memory Allocations**
  - OCC + MVCC

Acknowledgement: Prof. Andy Pavlo, CMU

LOCK THRASHING

- Each txn waits longer to acquire locks, causing other txn to wait longer to acquire locks.

- Can measure this phenomenon by removing deadlock detection/prevention overhead.
  - Force txns to acquire locks in primary key order.
  - Deadlocks are not possible.

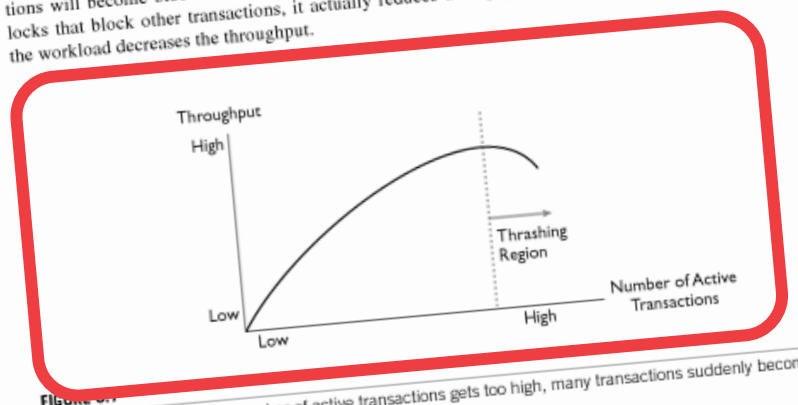
Acknowledgement: Prof. Andy Pavlo, CMU

converts the update lock to a write lock. This lock conversion can't lead to a lock conversion deadlock, because at most one transaction can have an update lock on the data item. (Two transactions must try to convert the lock at the same time to create a lock conversion deadlock.) On the other hand, the benefit of this approach is that an update lock does not block other transactions that read without expecting to update later on. The weakness is that the request to convert the update lock to a write lock may be delayed by other read locks. If a large number of data items are read and only a few of them are updated, the tradeoff is worthwhile. This approach is used in Microsoft SQL Server. SQL Server also allows update locks to be obtained in a SELECT (i.e., read) statement, but in this case, it will not downgrade the update locks to read locks, since it doesn't know when it is safe to do so.

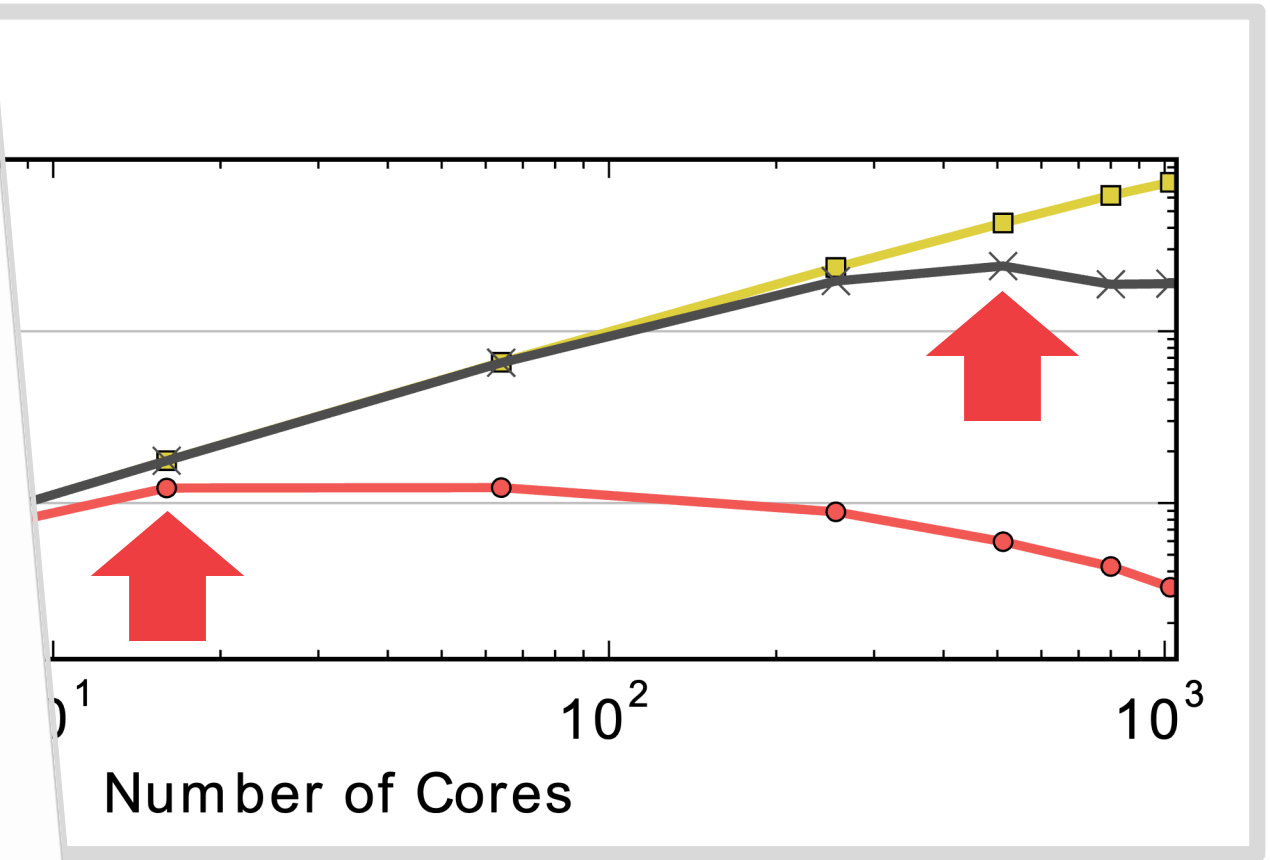
Lock Thrashing

By reducing the frequency of lock conversion deadlocks, we have dispensed with deadlock as a major performance consideration, so we are left with blocking situations. Blocking affects performance in a rather dramatic way. Until lock usage reaches a saturation point, it introduces only modest delays—significant, but not a serious problem. At some point, when too many transactions request locks, a large number of transactions suddenly become blocked, and few transactions can make progress. Thus, transaction throughput stops growing. Surprisingly, if enough transactions are initiated, throughput actually decreases. This is called **lock thrashing** (see Figure 6.7). The main issue in locking performance is to maximize throughput without reaching the point where thrashing occurs.

One way to understand lock thrashing is to consider the effect of slowly increasing the **transaction load**, which is measured by the number of active transactions. When the system is idle, the first transaction to run cannot block due to locks, because it's the only one requesting locks. As the number of active transactions grows, each successive transaction has a higher probability of becoming blocked due to transactions already running. When the number of active transactions is high enough, the next transaction to be started has virtually no chance of running to completion without blocking for some lock. Worse, it probably will get some locks before encountering one that blocks it, and these locks contribute to the likelihood that other active transactions will become blocked. So, not only does it not contribute to increased throughput, but by getting some locks that block other transactions, it actually reduces throughput. This leads to thrashing, where increasing the workload decreases the throughput.



Lock Thrashing. When the number of active transactions gets too high, many transactions suddenly become blocked, and few transactions can make progress.



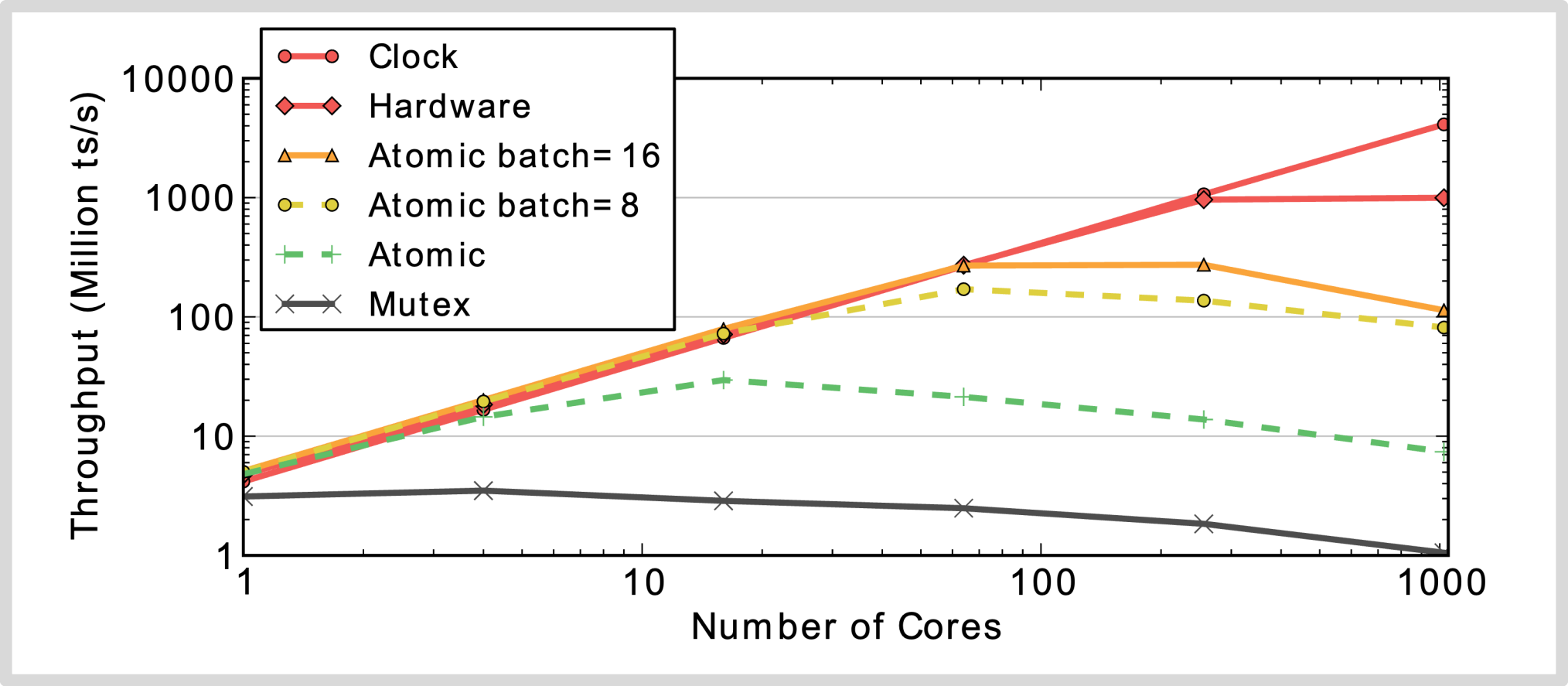
Acknowledgement: Prof. Andy Pavlo, CMU

TIMESTAMP ALLOCATION

- **Mutex**
  - Worst option.
- **Atomic Addition**
  - Requires cache invalidation on write.
- **Batched Atomic Addition**
  - Needs a back-off mechanism to prevent fast burn.
- **Hardware Clock**
  - Not sure if it will exist in future CPUs.
- **Hardware Counter**
  - Not implemented in existing CPUs.

Acknowledgement: Prof. Andy Pavlo, CMU

TIMESTAMP ALLOCATION



Acknowledgement: Prof. Andy Pavlo, CMU

MEMORY ALLOCATIONS

- Copying data on every read/write access slows down the DBMS because of contention on the memory controller.
  - In-place updates and non-copying reads are not affected as much.

- Default libc `malloc` is slow. Never use it.
  - We will discuss this further later in the semester.

Acknowledgement: Prof. Andy Pavlo, CMU

NEXT CLASS

- Multi-Version Concurrency Control