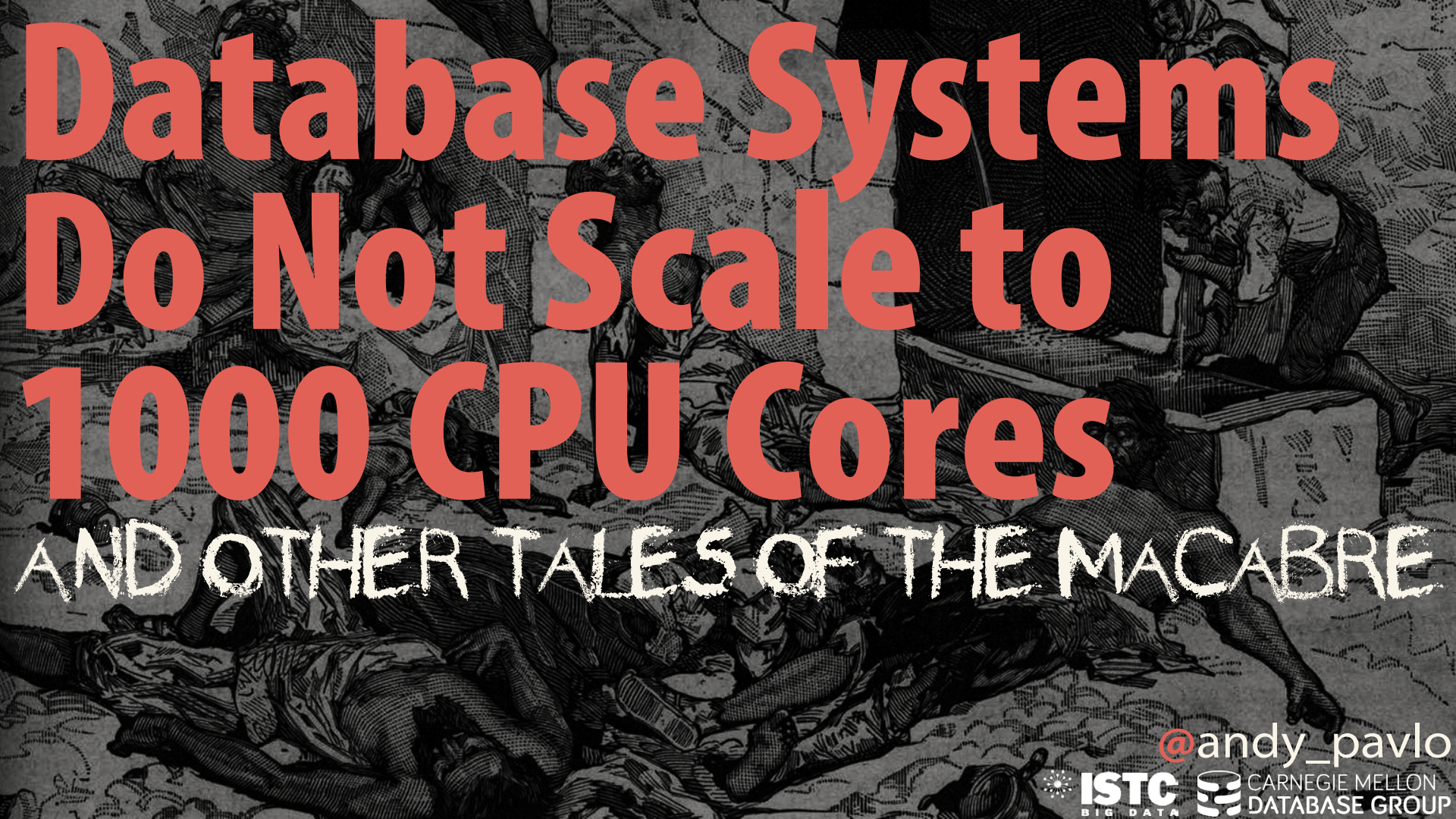

Database Systems
Do Not Scale to
1000 CPU Cores
AND OTHER TALES OF THE MACABRE
@andy_pavlo
ISTC
BIG DATA
CARNEGIE MELLON
DATABASE GROUP

Three million children die per year due to poor nutrition.

Source: http://www.wfp.org/hunger/stats

Three days after you die, stomach enzymes start to digest you.

Source: http://discovermagazine.com/2006/sep/10-20thingsdeath

Everyone in this room will be dead in 65 years.

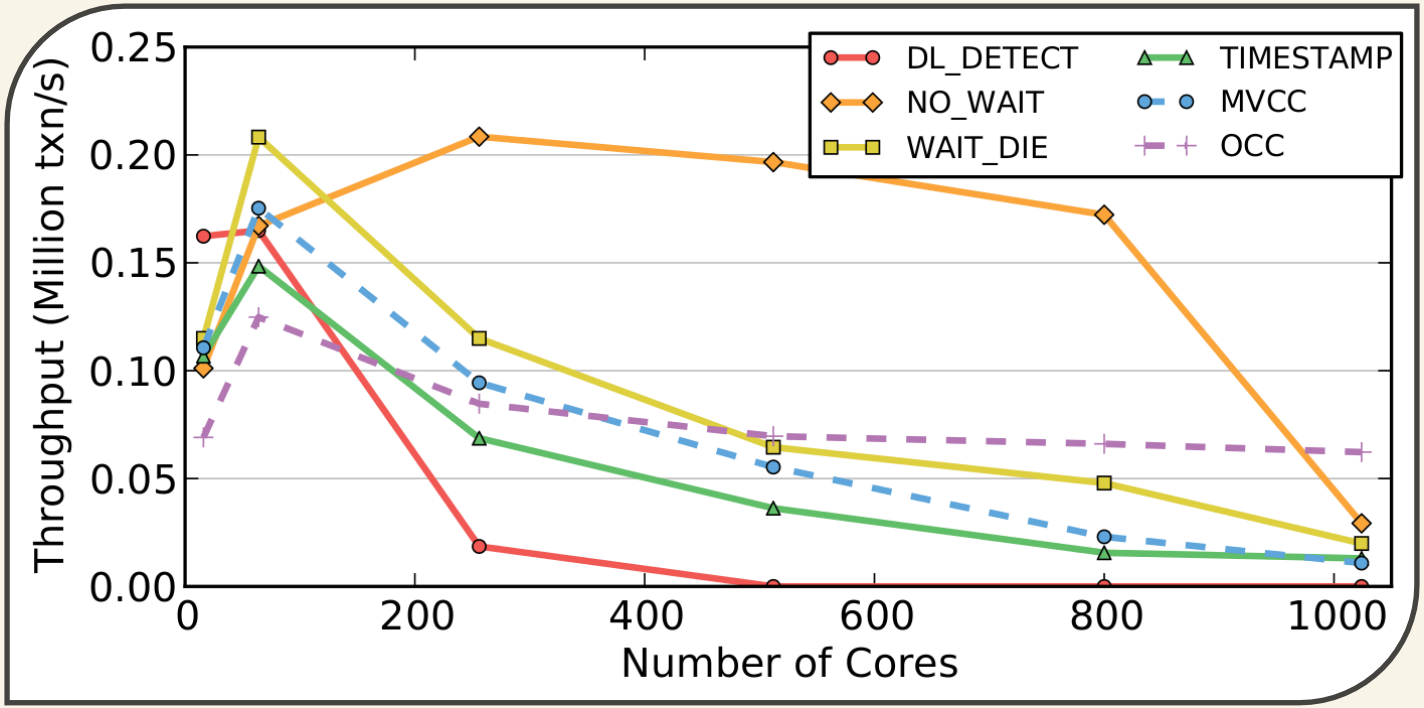
Source: http://discovermagazine.com/2006/sep/10-20thingsdeath

Database systems cannot scale to 1000 CPU cores.

Source: http://www.vldb.org/pvldb/vol8/p209-yu.pdf

YCSB //

DBx1000 on Graphite Simulator
Write-Intensive Workload
High Contention

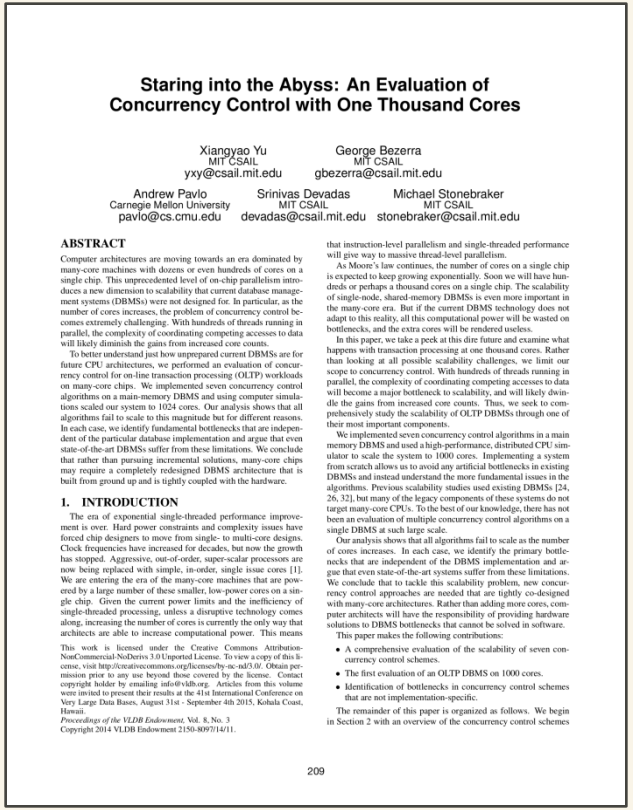


Why This Matters

- The era of single-core CPU speed-up is over.
- Database applications are getting more complex and larger.
- Existing DBMSs are unable to take advantage of future "many-core" CPU architectures.

Today's Talk

- Transaction Processing
- Experimental Platform
- Evaluation & Discussion
- The (Dire) Future



Staring into the Abyss: An Evaluation of Concurrency Control with One Thousand Cores

Xiangyao Yu
MIT CSAIL
yxy@csail.mit.edu

George Bezerra
MIT CSAIL
gbezerra@csail.mit.edu

Andrew Pavlo
Carnegie Mellon University
pavlo@cs.cmu.edu

Srinivas Devadas
MIT CSAIL
devadas@csail.mit.edu

Michael Stonebraker
MIT CSAIL
stonebraker@csail.mit.edu

ABSTRACT

Computer architectures are moving towards an era dominated by many-core machines with dozens or even hundreds of cores on a single chip. This unprecedented level of on-chip parallelism introduces a new dimension to scalability that current database management systems (DBMSs) were not designed for. In particular, as the number of cores increases, the problem of concurrency control becomes extremely challenging. With hundreds of threads running in parallel, the complexity of coordinating competing accesses to data will likely diminish the gains from increased core counts.

To better understand just how unprepared current DBMSs are for future CPU architectures, we performed an evaluation of concurrency control for on-line transaction processing (OLTP) workloads on many-core chips. We implemented seven concurrency control algorithms on a main-memory DBMS and using computer simulations scaled our system to 1024 cores. Our analysis shows that all algorithms fail to scale to this magnitude but for different reasons. In each case, we identify fundamental bottlenecks that are independent of the particular database implementation and argue that even state-of-the-art DBMSs suffer from these limitations. We conclude that rather than pursuing incremental solutions, many-core chips may require a completely redesigned DBMS architecture that is built from ground up and is tightly coupled with the hardware.

1. INTRODUCTION

The era of exponential single-threaded performance improvement is over. Hard power constraints and complexity issues have forced chip designers to move from single- to multi-core designs. Clock frequencies have increased for decades, but now the growth has stopped. Aggressive, out-of-order, super-scalar processors are now being replaced with simple, in-order, single issue cores [1]. We are entering the era of the many-core machines that are powered by a large number of these smaller, low-power cores on a single chip. Given the current power limits and the inefficiency of single-threaded processing, unless a disruptive technology comes along, increasing the number of cores is currently the only way that architects are able to increase computational power. This means that instruction-level parallelism and single-threaded performance will give way to massive thread-level parallelism.

As Moore's law continues, the number of cores on a single chip is expected to keep growing exponentially. Soon we will have hundreds or perhaps a thousand cores on a single chip. The scalability of single-node, shared-memory DBMSs is even more important in the many-core era. But if the current DBMS technology does not adapt to this reality, all this computational power will be wasted on bottlenecks, and the extra cores will be rendered useless.

In this paper, we take a peek at this dire future and examine what happens with transaction processing at one thousand cores. Rather than looking at all possible scalability challenges, we limit our scope to concurrency control. With hundreds of threads running in parallel, the complexity of coordinating competing accesses to data will become a major bottleneck to scalability, and will likely dwindle the gains from increased core counts. Thus, we seek to comprehensively study the scalability of OLTP DBMSs through one of their most important components.

We implemented seven concurrency control algorithms in a main memory DBMS and used a high-performance, distributed CPU simulator to scale the system to 1000 cores. Implementing a system from scratch allows us to avoid any artificial bottlenecks in existing DBMSs and instead understand the more fundamental issues in the algorithms. Previous scalability studies used existing DBMSs [24, 26, 32], but many of the legacy components of these systems do not target many-core CPUs. To the best of our knowledge, there has not been an evaluation of multiple concurrency control algorithms on a single DBMS at such large scale.

Our analysis shows that all algorithms fail to scale as the number of cores increases. In each case, we identify the primary bottlenecks that are independent of the DBMS implementation and argue that even state-of-the-art systems suffer from these limitations. We conclude that to tackle this scalability problem, new concurrency control approaches are needed that are tightly co-designed with many-core architectures. Rather than adding more cores, computer architects will have the responsibility of providing hardware solutions to DBMS bottlenecks that cannot be solved in software.

This paper makes the following contributions:

- A comprehensive evaluation of the scalability of seven concurrency control schemes.
- The first evaluation of an OLTP DBMS on 1000 cores.
- Identification of bottlenecks in concurrency control schemes that are not implementation-specific.

The remainder of this paper is organized as follows. We begin in Section 2 with an overview of the concurrency control schemes

Proceedings of the VLDB Endowment, Vol. 8, No. 3

Transaction Processing

On-line Transaction Processing

- Fast operations that ingest new data and then update state using ACID transactions.
- Transaction Example:
  - ***Send $50 from user A to user B***

Concurrency Control

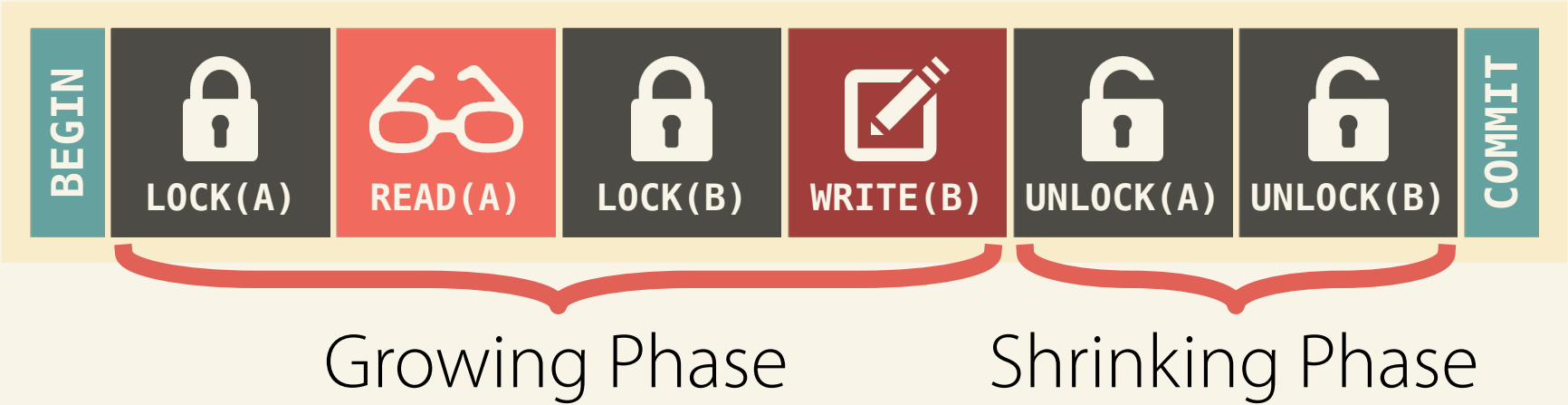
- Allows transactions to access a database in a multi-programmed fashion while preserving the illusion that each of them is executing alone on a dedicated system.
- Provides **A**tomicity + **I**solation in ACID

Concurrency Control

- **Two-Phase Locking** (Pessimistic)
- **Timestamp Ordering** (Optimistic)

Two-Phase Locking (2PL)

Transaction #1

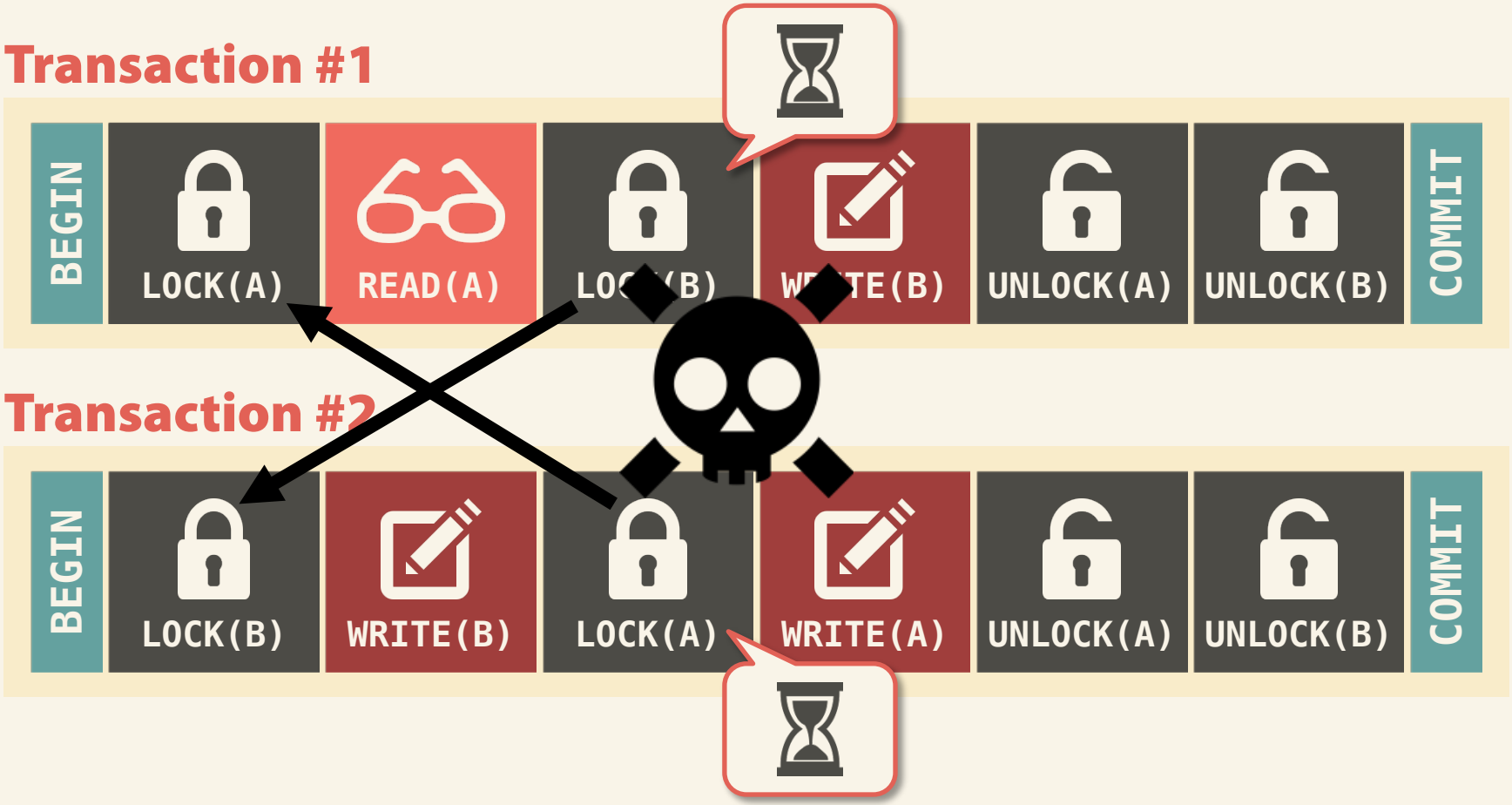


Two-Phase Locking (2PL)

Transaction #1

Transaction #2

Two-Phase Locking (2PL)



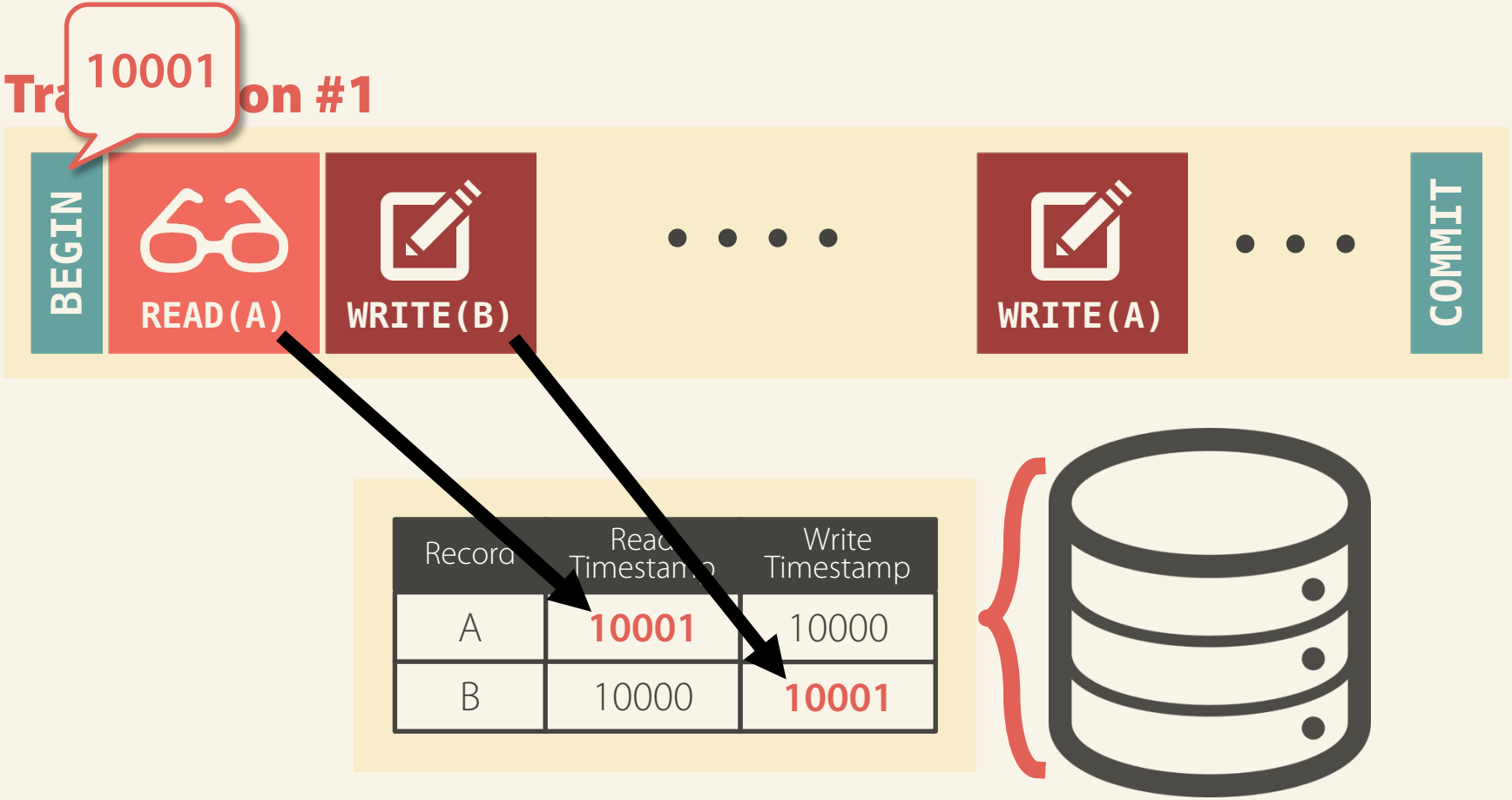
Two-Phase Locking (2PL)

- Deadlock Detection (**DEADLOCK**)
- Non-waiting Deadlock Prevention (**NO_WAIT**)
- Wait-and-Die Deadlock Prevention (**WAIT_DIE**)

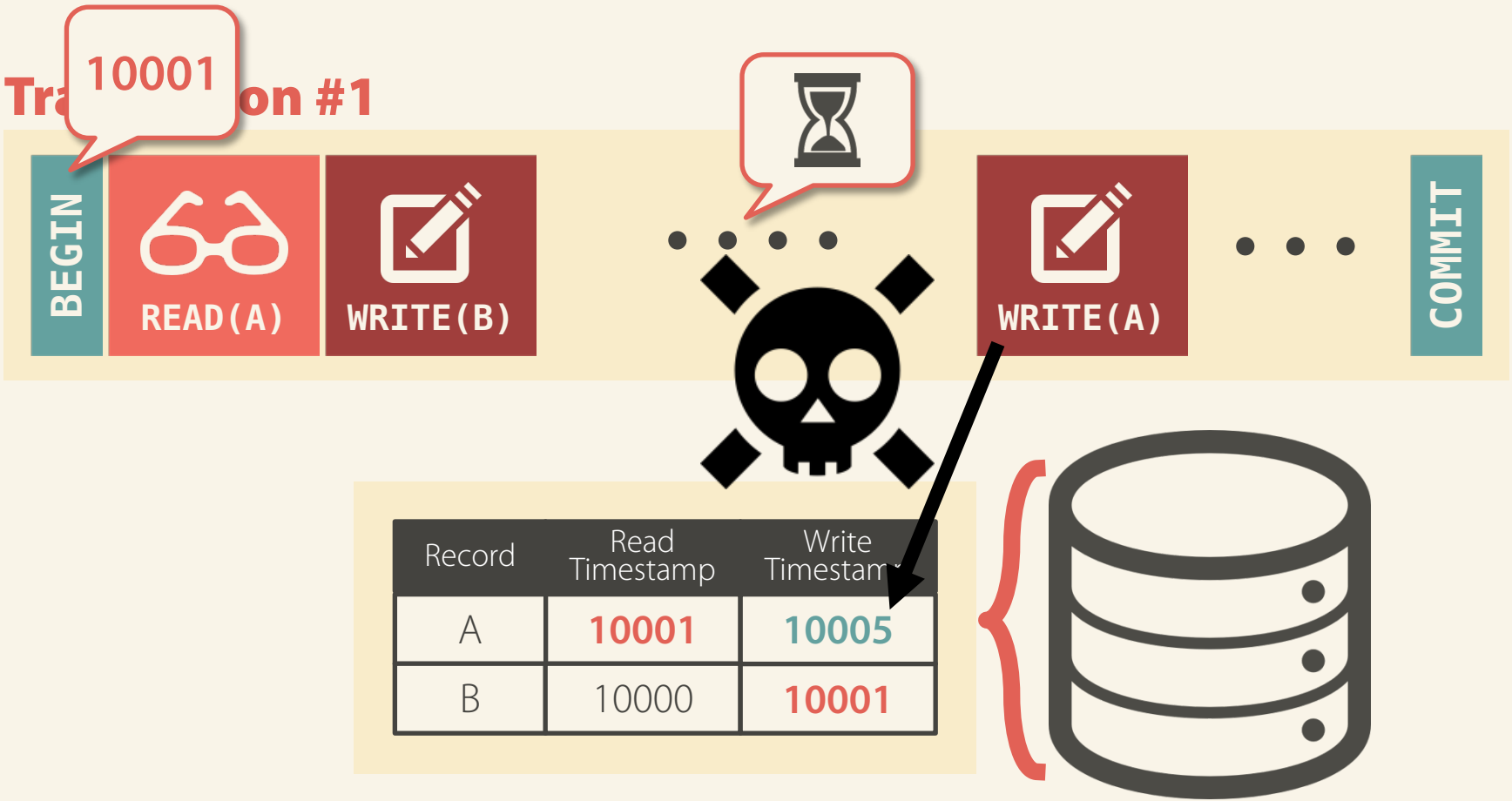
Timestamp Ordering (T/O)

Record	Read Timestamp	Write Timestamp
A	10000	10000
B	10000	10000

Timestamp Ordering (T/O)



Timestamp Ordering (T/O)



Timestamp Ordering (T/O)

- Basic T/O (**TIMESTAMP**)
- Multi-Version Concurrency Control (**MVCC**)
- Optimistic Concurrency Control (**OCC**)

Concurrency Control Schemes

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

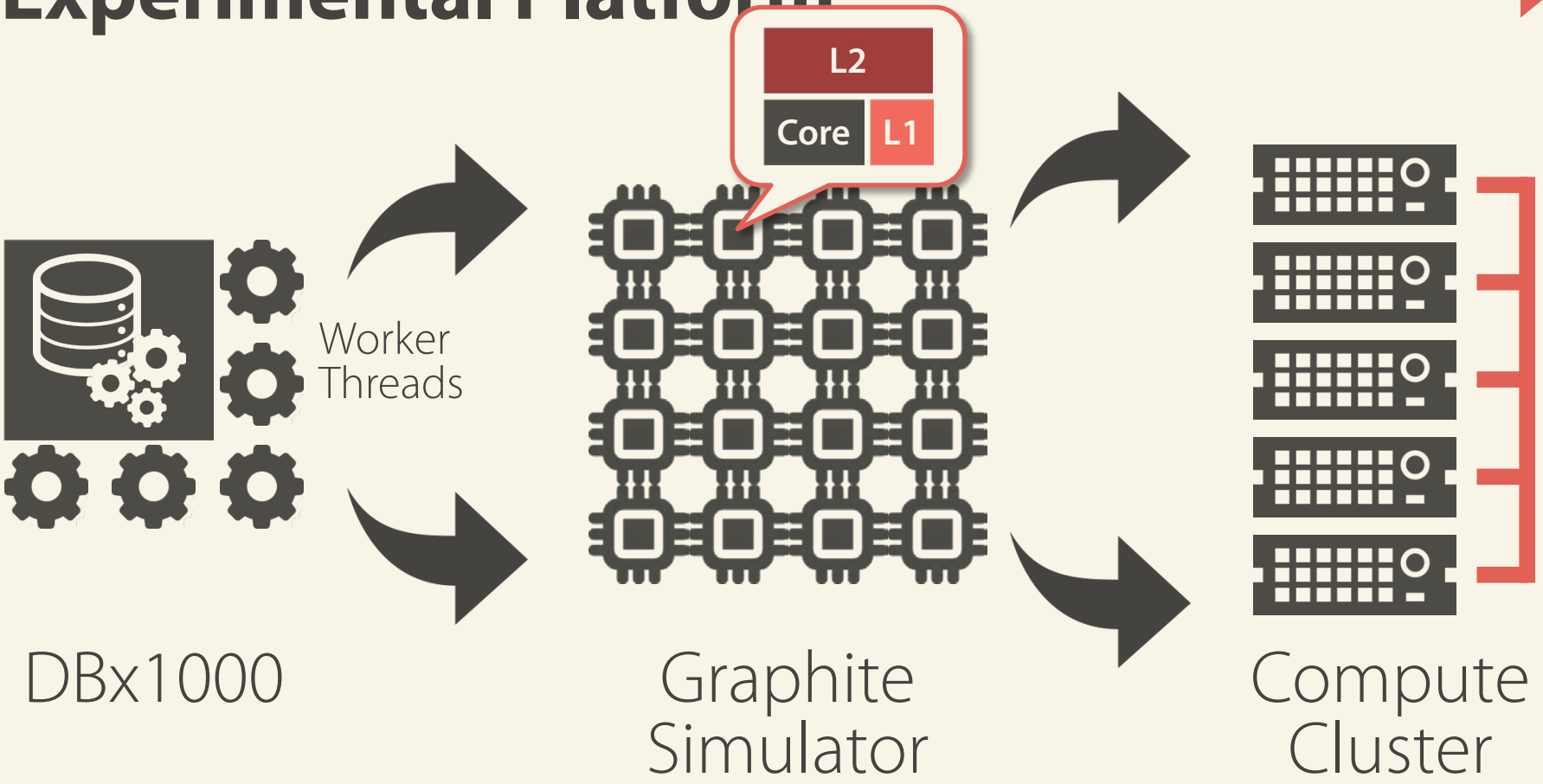
TIMESTAMP	Basic T/O Algorithm
MVCC	Multi-Version T/O
OCC	Optimistic Concurrency Control

Evaluation Testbed

No DBMS supports multiple CC schemes.

No CPU supports 1000 cores.

Experimental Platform



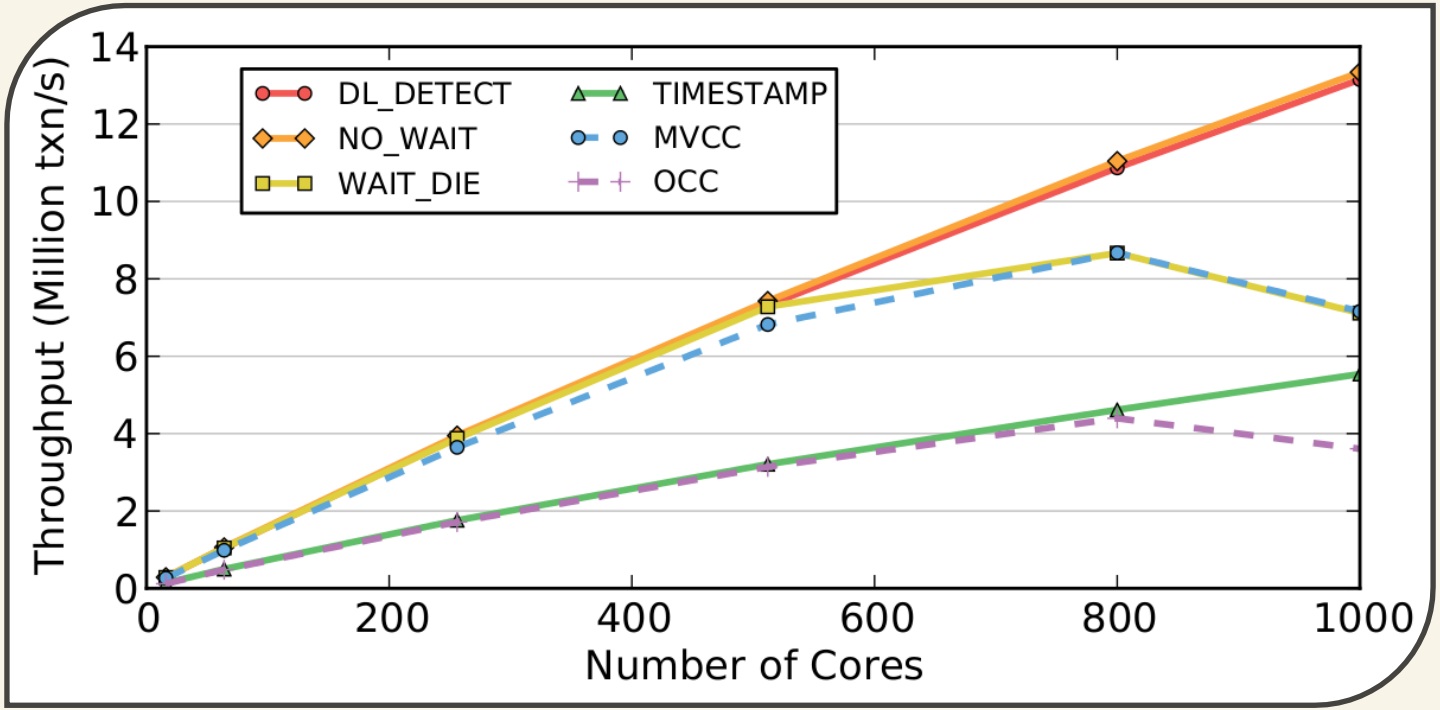
Target Workload

- 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.

Evaluation

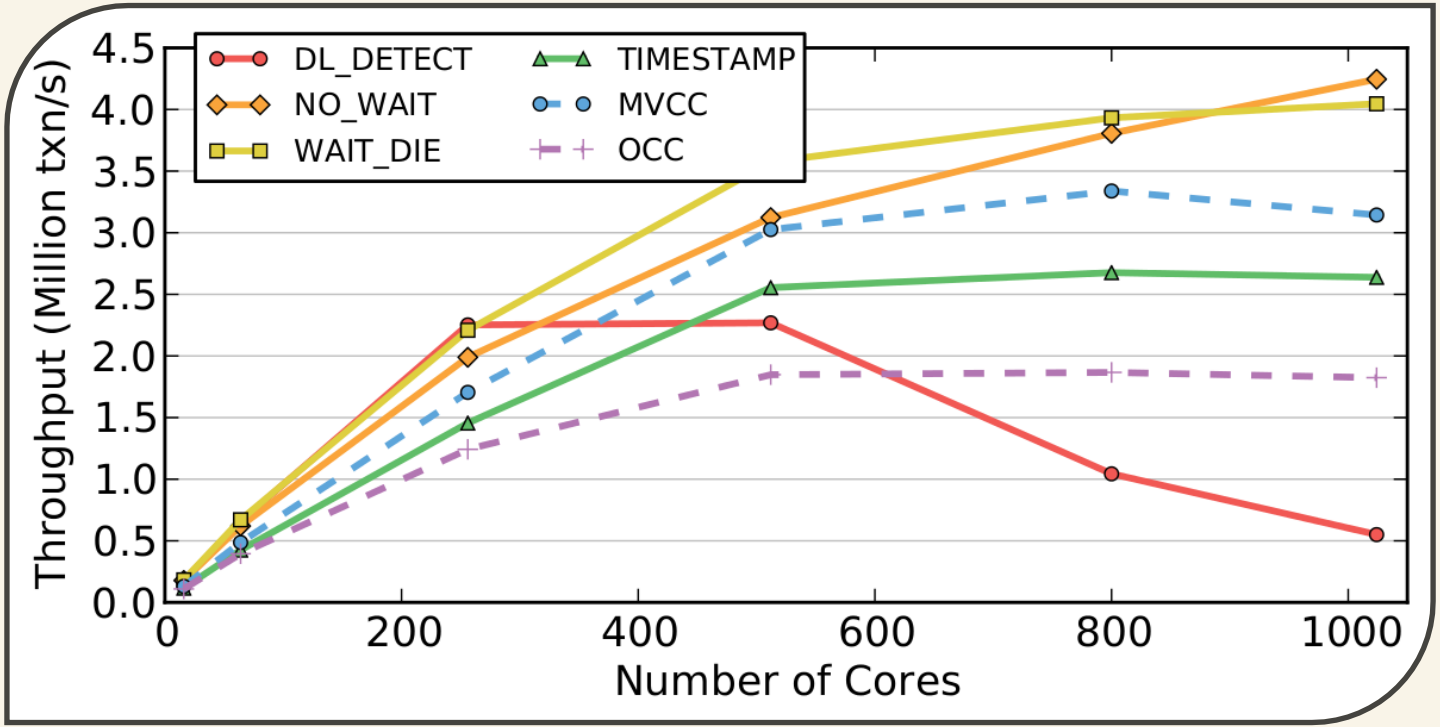
YCSB //

DBx1000 on Graphite Simulator
Read-Only Workload
No Contention



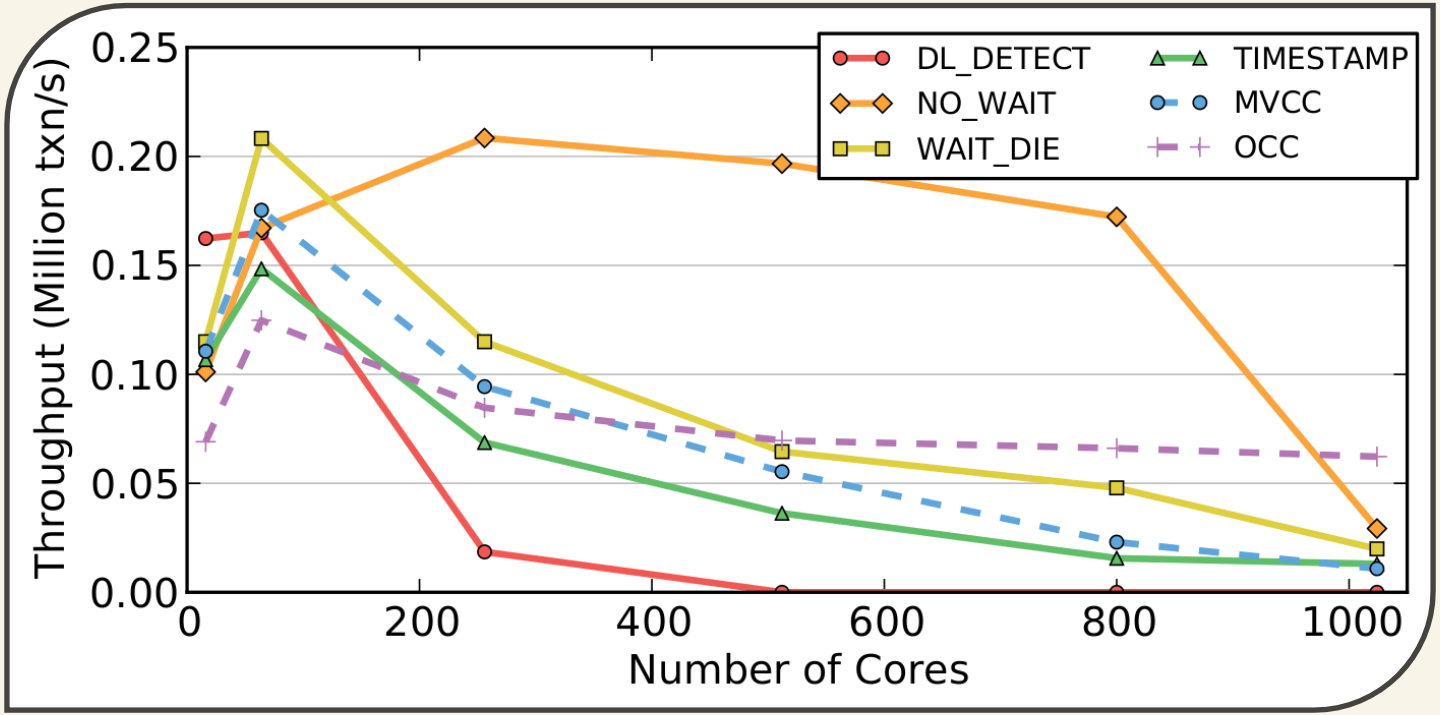
YCSB //

DBx1000 on Graphite Simulator
Write-Intensive Workload
Medium Contention



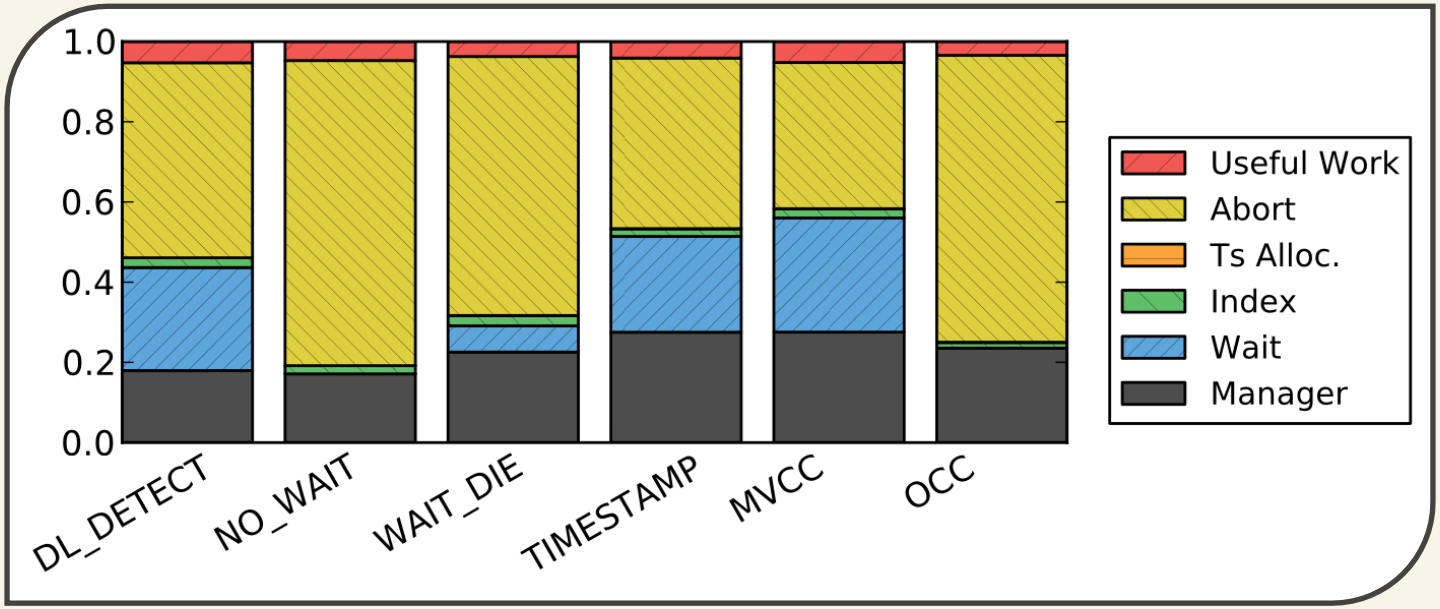
YCSB //

DBx1000 on Graphite Simulator
Write-Intensive Workload
High Contention



YCSB //

DBx1000 on Graphite Simulator
Write-Intensive Workload
High Contention



Time % Breakdown (512 Cores)

Bottlenecks

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

Bottlenecks

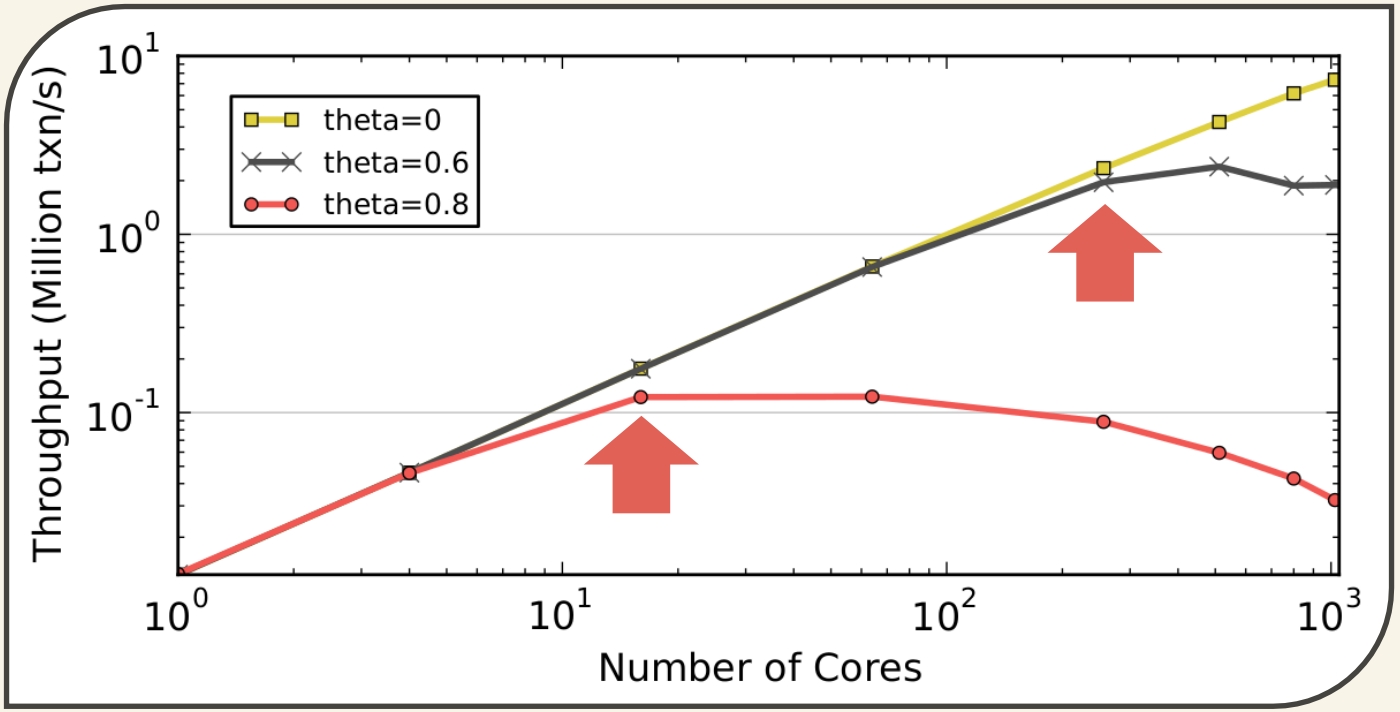
- Lock Thrashing
  - *DL_DETECT, WAIT_DIE*
- Timestamp Allocation
  - *All T/O algorithms + WAIT_DIE*
- Memory Allocations
  - *OCC + MVCC*

Locking Thrashing

- Each transaction waits longer to acquire locks, causing other transactions to wait a longer to acquire locks.
- The perfect workload is where transactions acquire locks in primary key order.

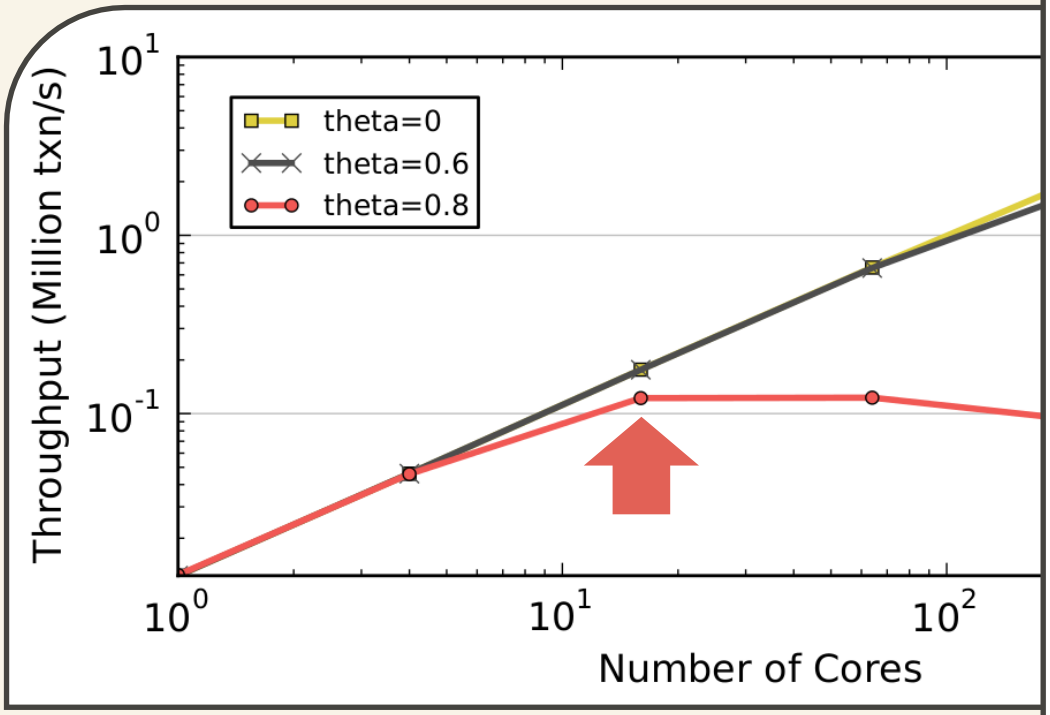
YCSB //

DBx1000 with 2PL DL_DETECT
Write-Intensive Workload
No Deadlocks (Ordered Lock Acquisition)



YCSB //

DBx1000 with 2PL DL_DETECT
Write-Intensive Workload
No Deadlocks (Ordered Lock Acquisition)



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

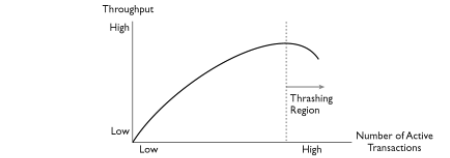


FIGURE 6.7

Potential Solutions

Hardware/Software Co-Design

- Bottlenecks can only be overcome through new hardware-level optimizations:
  - ***Hardware-accelerated Lock Sharing***
  - ***Asynchronous Memory Copying***
  - ***Decentralized Memory Controller.***

Next Steps

- Evaluating other main bottlenecks in DBMSs:
  - ***Logging + Recovery***
  - ***Indexes***
- Extend DBx1000 to support distributed concurrency control algorithms.

CARNEGIE MELLON DATABASE GROUP

MIT

ISTC BIG DATA

Xiangyao **Yu**

Andy **Pavlo**

Mike **Stonebraker**

Srini **Devadas**

http://cmudb.io/1000cores

END

@andy_pavlo