An Evaluation of Concurrency Control with One Thousand Cores

by

Xiangyao Yu

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2015

Signature redacted

Author .....
Department of Electrical Engineering and Computer Science
January 21, 2015

Signature redacted

Certified by .....
Professor Srinivas Devadas
Thesis Supervisor

Signature redacted

Accepted by .....
Professor Leslie A. Kolodziejski
Chair, Department Committee on Graduate Theses

An Evaluation of Concurrency Control with One Thousand Cores

by

Xiangyao Yu

Submitted to the Department of Electrical Engineering and Computer Science
on January 21, 2015, in partial fulfillment of the
requirements for the degree of
Master of Science in Computer Science and Engineering

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.

Thesis Supervisor: Professor Srinivas Devadas

Acknowledgments

First and foremost, I would like to thank my advisor, Srini Devadas. The atmosphere of freedom in the group was the key that nourished this thesis project and other exciting and crazy ideas. I appreciate your trust and support in the project even though it diverts from the mainstream research in our group. Thanks to your high expectations in research and aspiring guidance which have been driving me forward for the past two years.

I would like to thank my collaborators for their valuable advice and help. Thanks to George Bezzera for making this project possible and closely working besides me along the way. My graduate school experience would be much more bumpy if you were not here. Thanks to Andy Pavlo for leading the project, your encouragement and passion ignited my interest in database research. Thanks to Mike Stonebraker, I admire your wisdom and foresight which led the project into the right direction.

I thank all the members of HORNET group. It has been enjoyable to colaborate with Christopher Flether, Ling Ren, Albert Kwon and Marten van Dijk on security projects in my first year. I thank George Kurian and Omer Khan for brainstorming ideas together and helping me with the simulation environment. Thanks everyone for your insightful presentation and fruitful discussion during the group meetings.

Finally, I would like to thank my family. Thanks to my parents for their support in the past 25 years. They gave me the best environment to grow up freely. Thanks to my wife, Lei. My life becomes colorful when you entered it. Your love is forever the source of power in my life.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The era of single threaded performance improvement is over. The CPU frequency stops increasing due to power constraint. With today's semiconductor technology, the only plausible way to extend Moore's law is by exponentially increasing the number of cores on a single chip. It is already common today to see systems with upto 100 cores on a single chip and 1000-core CPUs are coming in the near future.

How to exploit such level of parallelism becomes a important but challenging problem for both the computer architecture and software applications. Traditional applications may experience all sorts of bottlenecks if run on multicore platforms unchanged. These bottlenecks can be either software bottlenecks like badly implemented synchronization primitives or critical sections, or hardware bottlenecks like memory bandwidth or communication latency. For future multicore processors, hardware designers and software programmers need to work closely together in order to solve these bottlenecks.

In this thesis, we will dive into one important class of applications to study the scalability problem. Specifically, we will look at the scalability of concurrency control in database applications. As a key component of a database, concurrency control guarantees the correctness of transaction processing. It is inherently hard to be parallelized due to the complicated dependency and communication patterns. And it encounters all sorts of scalability bottlenecks at 1000 cores.

1.1 Multicore Scalability

For decades since computers exist, CPU performance has been improving exponentially. The underlying impetus has been Moore's law which claims that the number of transistor on a single chip doubles every 24 months. At the same time, transistors also become smaller and faster, leading to higher clock frequency which leads to better performance.

However, since the last decade, due to the excessive power consumption as the transistor size scales down, the clock frequency of CPUs stops increasing. Computer architectures thus started to move into the multicore era where the performance improvement is driven by the increase of the number of cores on a single chip.

Different from a traditional parallel system with multiple servers, a multicore system features fast interconnection between cores since the on-chip communication is much faster than inter-machine communication. This makes it possible to apply multicore specific optimizations to applications.

1.2 Database Management Systems

Database Management Systems (DBMS) are especially designed software applications to interact with large amount of data. External users send queries to the database in the form of transactions. For example, a transaction may transfer money from one bank account to another one. A transaction is the unit work that must be done atomically. A transaction either fully commits or fully aborts. Partial transactions are not acceptable.

Transactions are considered to be correctly executed if they are serializable. Namely, the outcome of the database is equivalent to the outcome where transactions are executed in a certain sequential order. In practice, however, all DBMSs execute multiple transactions simultaneously in order to improve concurrency. Concurrency control algorithm is the key component in the DBMS to gurantee serializability while transactions are actually run in parallel.

As the number of cores increases, it becomes a challenge to scale the concurrency control algorithms. Many traditional concurrency control schemes require centralized data structures or critical sections which are inherently unscalable. There are also artificial bottlenecks in existing DBMS implementations since they were not optimized for multicore processors. Several efforts have been made in recent years to scale concurrency control. Some are optimizations based on existing systems and others are completely new algorithms developed from scratch. Yet none of these work have been tested on 1000 cores. At this level of parallelism, concurrency control will reach some fundamental bottlenecks which none of these previous algorithms would be able to solve.

1.3 Contribution

In main goal of this thesis is to understand the fundamental scalability bottlenecks in existing concurrency control algorithms. We implemented all the sever classic concurrency control algorithms from scratch. Several scalability optimizations are implemented to remove as many software bottlenecks as possible. The system is then scaled upto one thousand cores under different workloads. To our knowledge, this is the first work that does extensive evaluations of all the seven concurrency control on 1000-core systems.

Our experiments showed that none of the seven concurrency control algorithms scales on 1000 cores. But they all have different reasons of failure. In this thesis, we will identify these fundamental scalability bottlenecks that are not implementation specific but inherent to the concurrency control algorithms themselves.

Both software and hardware solutions are proposed to solve some of the scalability bottlenecks. The tradeoff between these different algorithms are also compared. We believe a software/hardware codesign approach is the right path to solve the DBMS scalability problem.

1.4 Thesis Organization

The thesis is organized as follows. Chapter 2 gives background on concurrency control algorthms. Chapter 3 introduces the many-core test bed used for our experiments. Chapter 4 discusses the design choices and optimizations applied to our DBMS. We present evaluation results in Chapter 5 and discuss the implevation in Chapter 6. Related work is discussed in Chapter 7 and finally Chapter 8 concludes the paper.

Chapter 2

Concurrency Control

OLTP database systems support the part of an application that interacts with *end-users*; for example, it is the system that processes new orders, responds to a page request, or performs a financial transaction. End-users interact with the front-end application by sending it requests to perform some function (e.g., reserve a seat on a flight). The application processes these requests and then executes transactions in the DBMS. Such users could be humans on their personal computer or mobile device, or another computer program potentially running somewhere else in the world.

A *transaction* in the context of one of these systems is the execution of a sequence of one or more operations (e.g., SQL queries) on a shared database to perform some higher-level function [18]. It is the basic unit of change in a DBMS: partial transactions are not allowed, and the effect of a group of transactions on the database's state is equivalent to any serial execution of all transactions. The transactions in modern OLTP workloads have three salient characteristics: (1) they are short-lived (i.e., no user stalls), (2) they touch a small subset of data using index look-ups (i.e., no full table scans or large joins), and (3) they are repetitive (i.e., executing the same queries with different inputs) [39].

An OLTP DBMS is expected to maintain four properties for each transaction that it executes: (1) atomicity, (2) consistency, (3) isolation, and (4) durability. These unifying concepts are collectively referred to with the *ACID* acronym [21]. Concurrency control permits end-users to access a database in a multi-programmed

fashion while preserving the illusion that each of them is executing their transaction alone on a dedicated system [4]. It essentially provides the atomicity and isolation guarantees in the system.

We now describe the different concurrency control schemes that we explored in our many-core evaluation. For this discussion, we follow the canonical categorization that all concurrency schemes are either a variant of *two-phase locking* or *timestamp ordering* protocols [4]. Table 2.1 provides a summary of these different schemes.

2.1 Two-Phase Locking

Two-phase locking (2PL) was the first provably correct method of ensuring the correct execution of concurrent transactions in a database system [7, 13]. Under this scheme, transactions have to acquire locks for a particular element in the database before they are allowed to execute a read or write operation on that element. The transaction must acquire a *read lock* before it is allowed to read that element, and similarly it must acquire a *write lock* in order to modify that element. The DBMS maintains locks for either each tuple or at a higher logical level (e.g., tables, partitions) [15].

The ownership of locks is governed by two rules: (1) different transactions cannot simultaneously own *conflicting locks*, and (2) once a transaction surrenders ownership of a lock, it may never obtain additional locks [4]. A read lock on an element conflicts with a write lock on that same element. Likewise, a write lock on an element conflicts with a write lock on the same element.

In the first phase of 2PL, known as the *growing phase*, the transaction is allowed to acquire as many locks as it needs without releasing locks [13]. When the transaction releases a lock, it enters the second phase, known as the *shrinking phase*; it is prohibited from obtaining additional locks at this point. When the transaction terminates (either by committing or aborting), all the remaining locks are automatically released back to the coordinator.

2PL is considered a pessimistic approach in that it assumes that transactions will conflict and thus they need to acquire locks to avoid this problem. If a transaction

2PL	**DL_DETECT**	2PL with deadlock detection.
	NO_WAIT	2PL with non-waiting deadlock prevention.
	WAIT_DIE	2PL with wait-and-die deadlock prevention.
T/O	**TIMESTAMP**	Basic T/O algorithm.
	MVCC	Multi-version T/O.
	OCC	Optimistic concurrency control.
	H-STORE	T/O with partition-level locking.

Table 2.1: The concurrency control schemes evaluated in this paper

is unable to acquire a lock for an element, then it is forced to wait until the lock becomes available. If this waiting is uncontrolled (i.e., indefinite), then the DBMS can incur deadlocks [4]. Thus, a major difference among the different variants of 2PL is in how they handle deadlocks and the actions that they take when a deadlock is detected. We now describe the different versions of 2PL that we have implemented in our evaluation framework, contrasting them based on these two details:

2.1.1 2PL with Deadlock Detection (DL_DETECT)

In DL_DETECT, deadlocks are allowed to happen. The system detects the deadlock after it is formed and resolves it by aborting some transactions causing the deadlock.

The DBMS maintains a *waits-for* graph which tracks the dependency information between transactions. The waits-for graph is a directed acyclic graph (DAG) where a node represents a transaction and an edge represents the dependency between two transactions. If transaction A waits for another transaction B, an edge is added to the waits-for graph from node A to node B. If a cycle is formed in the waits-for graph, all the transactions in the cycle are waiting and cannot make forward progrss and a deadlock is generated.

In DL_DETECT, the DBMS monitors the *waits-for* graph and checks for cycles (i.e., deadlocks) [20]. When a deadlock is found, the system must choose a transaction within the cycle to abort and restart in order to break the cycle. In practice, the deadlock detector is usually a centralized structure. The detection can be periodic or only happen when the waiting time of a transaction exceeds a certain threshold. The detector chooses which transaction to abort based on the amount of resources it has

already used (e.g., the number of locks it holds) to minimize the cost of restarting a transaction [4].

2.1.2 2PL with Non-waiting Deadlock Prevention (NO_WAIT)

Unlike DL_DETECT where the DBMS waits to find deadlocks after they occur, NO_WAIT is more cautious in that a transaction is aborted when the system suspects that a deadlock might occur [4].

In NO_WAIT, when a lock request is denied due to lock type conflicts, the requesting transaction is immediately aborted (i.e., it is not allowed to wait to acquire the lock) and all the locks already acquired should be released.

NO_WAIT does not require a deadlock detector or any centralized data structure. In this sense, it is more scalable than DL_DETECT. However, the algorithm does suffer from livelock where a transaction may always abort with each restart. In the worse case, a certain transaction may take a long time to finish.

2.1.3 2PL with Waiting Deadlock Prevention (WAIT_DIE)

This is a variation of the NO_WAIT scheme allowing a transaction to wait in certain cases. In WAIT_DIE, each transaction is assigned a timestamp at its beginning which indicates the global order of the transaction. The timestamp should be unique for each transaction and monotonically increasing.

A transaction A is allowed to wait for another transaction B only if the timestamp of A is smaller than the timestamp of B, in other words, A is old than B. If the transaction A is younger, then it is aborted (hence the term "dies") and is forced to restart [4]. A restarting transaction still uses the previously assinged timestamp.

This scheme does not generate deadlocks since the edges in the waits-for graph can only point from old transactions to young transactions and thus no cycle can be formed. Unlike NO_WAIT, however, WAIT_DIE does not generate livelocks because the oldest transaction in the system can wait for every lock and thus eventuall finish.

2.2 Timestamp Ordering

Timestamp ordering (T/O) concurrency control schemes generate a serialization order of transactions a priori and then the DBMS enforces this order. A transaction is assigned a unique, monotonically increasing timestamp before it is executed; this timestamp is used by the DBMS to process conflicting operations in the proper order (e.g., read and write operations on the same element, or two separate write operations on the same element) [4].

We now describe the T/O schemes implemented in our test-bed. The key differences between the schemes are (1) the granularity that the DBMS checks for conflicts (i.e., tuples vs. partitions) and (2) when the DBMS checks for these conflicts (i.e., while the transaction is running or at the end).

2.2.1 Basic T/O (TIMESTAMP)

In TIMESTAMP, each tuple in the database has a read timestamp and a write timestamp associated with it. Every time a transaction reads or modifies a tuple in the database, the DBMS compares the timestamp of the transaction with the read and write timestamp of the tuple, which is the timestamp of the last transaction that reads or writes the same tuple.

For any read or write operation, the DBMS rejects the request if the transaction's timestamp is less than the write timestamp of the tuple. Because the value of the tuple is updated at a later logical time, it should not be observed at a previous logical time. For a write operation, the DBMS rejects it if the transaction's timestamp is less than the read timestamp to that tuple. This is because the read with a larger timestamp has already observed the current data, an update with a smaller timestamp would not be allowed. For each successful read or write request, the read or write timestamp of the tuple should also be properly updated.

In TIMESTAMP, a read query should make a local copy of the tuple to ensure repeatable reads since it is not protected by locks. When a transaction is aborted, it is assigned a new timestamp and then restarted. This corresponds to the "basic T/O"

algorithm as described in [4], but our implementation uses a decentralized scheduler.

2.2.2 Multi-version Concurrency Control (MVCC)

Under MVCC, every write operation creates a new version of a tuple in the database [5, 6]. Each version is tagged with the timestamp of the transaction that created it. The DBMS maintains an internal list of the versions of an element.

For a read operation, the DBMS determines which version in this list the transaction will access. An older version of the tuple may be returned depending on the timestamp of the requesting transaction. Compared to TIMESTAMP, MVCC does not reject operations that arrive late. That is, the DBMS does not reject a read operation because the element that it targets has already been overwritten by another transaction [6].

2.2.3 Optimistic Concurrency Control (OCC)

The DBMS tracks the read/write sets of each transaction and stores all of their write operations in their private workspace [29]. When a transaction commits, the system determines whether that transaction's read set overlaps with the write set of any concurrent transactions. If no overlap exists, then the DBMS applies the changes from the transaction's workspace into the database; otherwise, the transaction is aborted and restarted.

The advantage of this approach for main memory DBMSs is that transactions write their updates to shared memory only at commit time, and thus the contention period is short [43]. Modern implementations of OCC include Silo [43] and Microsoft's Hekaton [12, 30]. In this paper, our algorithm is similar to Hekaton in that we parallelize the validation phase and thus is more scalable than the original algorithm [29].

2.2.4 T/O with Partition-level Locking (H-STORE)

The database is divided into disjoint subsets of memory called *partitions*. Each partition is protected by a lock and is assigned a single-threaded execution engine that

has exclusive access to that partition. Only one transaction holds the lock for a partition at a time. Each transaction must acquire the locks for all of the partitions that it needs to access before it is allowed to start running. This requires the DBMS to know what partitions that each individual transaction will access before it begins [35]. When a transaction request arrives, the DBMS assigns it a timestamp and then adds it to all of the lock acquisition queues for its target partitions. The execution engine for a partition removes a transaction from the queue and grants it access to that partition if the transaction has the oldest timestamp in the queue [39]. Smallbase was an early proponent of this approach [23], and more recent examples include H-Store [28]. and its commercial implementation VoltDB [1].

Chapter 3

Many-Core DBMS Test-Bed

Since many-core chips do not yet exist, we performed our analysis through Graphite [31], a CPU simulator that can scale up to 1024 cores. For the DBMS, we implemented a main memory OLTP engine that only contains the functionality needed for our experiments. The motivation for using a custom DBMS is two fold. First, we can ensure that no other bottlenecks exist other than concurrency control. This allows us to study the fundamentals of each scheme in isolation without interference from unrelated features. Second, using a full-featured DBMS is impractical due to the considerable slowdown of simulators (e.g., Graphite has an average slowdown of 10,000×). Our engine allows us to limit the experiments to reasonable times. We now describe the simulation infrastructure, the DBMS engine, and the workloads used in this study.

3.1 Simulator and Target Architecture

Graphite [31] is a fast CPU simulator for large-scale multi-core systems. Graphite runs off-the-shelf Linux applications by creating a separate thread for each core in the architecture. As shown in Fig. 3-1, each application thread is attached to a simulated core thread that can then be mapped to different processes on separate host machines. For additional performance, Graphite relaxes cycle accuracy, using periodic synchronization mechanisms to model instruction-level granularity. As with other similar CPU simulators, it only executes the application and does not model

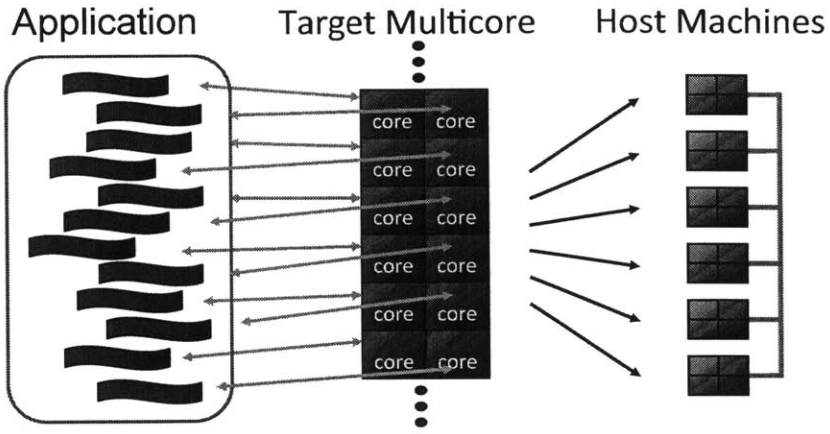


Figure 3-1: **Graphite Simulator Infrastructure** – Application threads are mapped to simulated core threads deployed on multiple host machines.

the operating system. For this paper, we deployed Graphite on a 22-node cluster, each with dual-socket Intel Xeon E5-2670 and 64GB of DRAM.

The target architecture is a tiled multi-core CPU, where each tile contains a low-power, in-order processing core, 32KB L1 instruction/data cache, a 512KB L2 cache slice, and a network router. This is similar to other commercial CPUs, such as Tilera's Tile64 (64 cores), Intel's SCC (48 cores), and Intel's Knights Landing (72 cores) [2]. Tiles are interconnected using a high-bandwidth, 2D-mesh on-chip network, where each hop takes two cycles. Both the tiles and network are clocked at 1GHz frequency. A schematic of the architecture for a 64-core machine is depicted in Fig. 3-2.

We use a shared L2-cache configuration because it is the most common last-level cache design for commercial multi-cores. In a comparison experiment between shared and private L2-caches, we observe that shared caches lead to significantly less memory traffic and higher performance for OLTP workloads due to its increased aggregate cache capacity (results not shown). Since L2 slices are distributed among the different tiles, the simulated multi-core system is a NUCA (Non-Uniform Cache Access) architecture, where L2-cache latency increases with distance in the 2D-mesh.

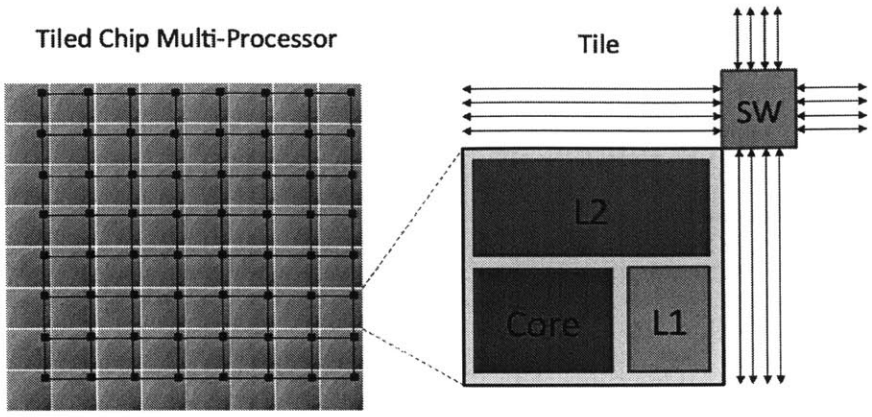


Figure 3-2: **Target Architecture** – Tiled chip multi-processor with 64 tiles and a 2D-mesh network-on-chip. Each tile contains a processing core, L1 and L2 caches, and a network switch (SW).

3.2 DBMS

We implemented our own lightweight main memory DBMS based on `pthreads` to run in Graphite. It executes as a single process with the number of worker threads equal to the number of cores, where each thread is mapped to a different core. All data in our DBMS is stored in memory in a row-oriented manner. The system supports basic hash table indexes and a pluggable lock manager that allows us swap in the different implementations of the concurrency control schemes described in Chapter 2. It also allows the indexes and lock manager to be partitioned (as in the case with the H-STORE scheme) or run in a centralized mode.

Client threads are not simulated in our system; instead, each worker contains a fixed-length queue of transactions that are served in order. This reduces the overhead of network protocols, which are inherently difficult to model in the simulator. Each transaction contains program logic intermixed with query invocations. The queries are executed serially by the transaction's worker thread as they are encountered in the program logic. Transaction statistics, such as throughput, latency, and abort rates, are collected after a warm-up period that is long enough for the system to achieve a steady state.

In addition to runtime statistics, our DBMS also reports how much time each

transaction spends in the different components of the system [22]. We group these measurements into six categories:

USEFUL WORK: The time that the transaction is actually executing application logic and operating on tuples in the system.

ABORT: The overhead incurred when the DBMS rolls back all of the changes made by a transaction that aborts.

TS ALLOCATION: The time that it takes for the system to acquire a unique timestamp from the centralized allocator. For those concurrency control schemes that require a timestamp, the allocation overhead happens only once per transaction.

INDEX: The time that the transaction spends in the hash indexes for tables, including the overhead of low-level latching of the buckets in the hash tables.

WAIT: The total amount of time that a transaction has to wait. A transaction may either wait for a lock (e.g., 2PL) or for a tuple whose value is not ready yet (e.g., T/O).

MANAGER: The time that the transaction spends in the lock manager or the timestamp manager. This excludes any time that it has to wait.

3.3 Workloads

We next describe the two benchmarks that we implemented in our test-bed for this analysis.

3.3.1 YCSB

The Yahoo! Cloud Serving Benchmark is a collection of workloads that are representative of large-scale services created by Internet-based companies [9]. For all of the YCSB experiments in this paper, we used a ~20GB YCSB database containing a single table with 20 million records. Each YCSB tuple has a single primary key column and then 10 additional columns each with 100 bytes of randomly generated string data. The DBMS creates a single hash index for the primary key.

Each transaction in the YCSB workload by default accesses 16 records in the database. Each access can be either a read or an update. The transactions do not perform any computation in their program logic. All of the queries are independent from each other; that is, the input of one query does not depend on the output of a previous query. The records accessed in YCSB follows a Zipfian distribution that is controlled by a parameter called *theta* that affects the level of contention in the benchmark [19]. When *theta*=0, all tuples are accessed with the same frequency. But when *theta*=0.6 or *theta*=0.8, a hotspot of 10% of the tuples in the database are accessed by ~40% and ~60% of all transactions, respectively.

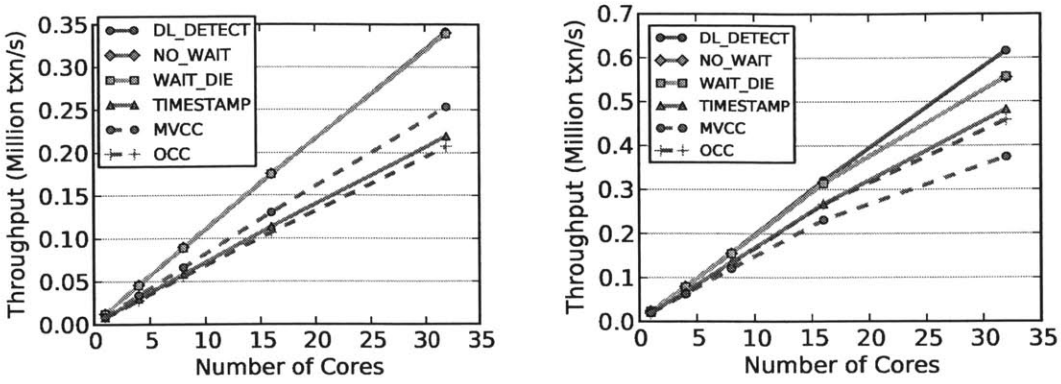
3.3.2 TPC-C

This benchmark is the current industry standard for evaluating the performance of OLTP systems [41]. It consists of nine tables that simulate a warehouse-centric order processing application. All of the transactions in TPC-C provide a `WAREHOUSE` id as an input parameter for the transaction, which is the ancestral foreign key for all tables except `ITEM`. For a concurrency control algorithm that requires data partitioning (i.e., H-STORE), TPC-C is partitioned based on this warehouse id.

Only two (`Payment` and `NewOrder`) out of the five transactions in TPC-C are modeled in our simulation. Since these two comprise 88% of the total TPC-C workload, this is a good approximation. Our version of TPC-C is a "good faith" implementation, although we omit the "thinking time" for worker threads. Each worker issues transactions without pausing; this mitigates the need to increase the size of the database with the number of concurrent transactions.

3.4 Simulator vs. Real Hardware

To show that using the Graphite simulator generates results that are comparable to existing hardware, we deployed our DBMS on an Intel Xeon E7-4830 and executed a read-intensive YCSB workload with medium contention (*theta*=0.6). We then executed the same workload in Graphite with the same number of cores.



(a) Graphite Simulation

(b) Real Hardware

Figure 3-3: **Simulator vs. Real Hardware** – Comparison of the concurrency control schemes running in Graphite and a real multi-core CPU using the YCSB workload with medium contention (*theta*=0.6).

The results in Fig. 3-3 show that all of the concurrency control schemes exhibit the same performance trends on Graphite and the real CPU. We note, however, that the relative performance difference between MVCC, TIMESTAMP, and OCC is different in Fig. 3-3b. This is because MVCC accesses memory more than the other two schemes and those accesses are more expensive on a two-socket system. Graphite models a single CPU socket and thus there is no inter-socket traffic. In addition to this, the throughput of the T/O-based and WAIT_DIE schemes drops on 32 cores due to the overhead of cross-core communication during timestamp allocation. We address this issue in Section 4.3.

Chapter 4

Design Choices and Optimizations

One of the main challenges of this study was designing a DBMS and concurrency control schemes that are as scalable as possible. When deploying a DBMS on 1000 cores, many secondary aspects of the implementation become a hindrance to performance. We did our best to optimize each algorithm, removing all possible scalability bottlenecks while preserving their essential functionality. Most of this work was to eliminate shared data structures and devise distributed versions of the classical algorithms [4].

In this section, we discuss our experience with developing a many-core OLTP DBMS and highlight the design choices we made to achieve a scalable system. Additionally, we identify fundamental bottlenecks of both the 2PL and T/O schemes and show how hardware support mitigates these problems. We present our detailed analysis of the individual schemes in Chapter 5.

4.1 General Optimizations

We first discuss the optimizations that we added to improve the DBMS's performance across all concurrency control schemes.

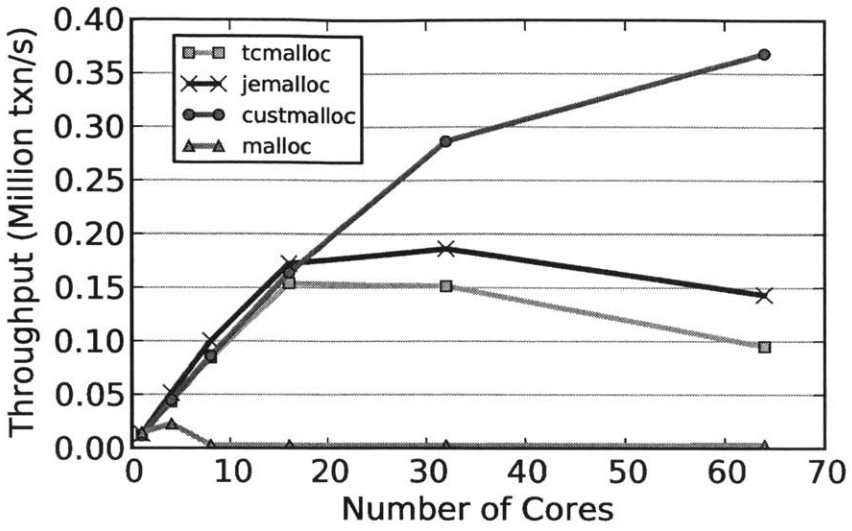


Figure 4-1: **Different Malloc** – Comparison of different malloc schemes.

4.1.1 Memory Allocation:

One of the first limitations we encountered when trying to scale our DBMS to large core counts was the `malloc` function. When using the default Linux version of `malloc`, we found that the DBMS spends most of the time waiting for memory allocation. This is a problem even for read-only workloads, since the DBMS still needs to copy records for reads in TIMESTAMP and to create internal meta-data handles for access tracking data structures. We tried running optimized versions (`TCMalloc` [16], `jemalloc` [14]), but both yielded similar disappointing results.

To better understand the problems that we had with different memory allocation libraries, we compared different malloc algorithms on a real 80-core server (Intel Xeon E7-8870). The results are shown in Fig. 4-1. It is clear that the default malloc function is not scalable at all. Tcmalloc and jemalloc are more scalable than the default malloc, but the performance still decreases after around 32 cores.

We believe the main reason that jemalloc and tcmalloc are not scalable is because of the local pool management. Both jemalloc and tcmalloc use per thread memory pool (local pool). Only when the local pool is empty or full, the global malloc function is called to allocate or free a large chunk of memory. At the number of cores increase,

the local pool size should also increase to avoid frequent accessing the global malloc. However, for both tcmalloc and jemalloc, this size is static. For tmalloc, the local pool size is 2MB, and for jemalloc, allocation is done in the unit of 2MB chunks. At high core count, such static local pool size will incur a large number of function calls to the global malloc, which limits the scalability.

We overcame this by writing a custom `malloc` implementation. Similar to `tcmalloc` and `jemalloc`, each thread is assigned its own memory pool. But the difference is that our allocator automatically resizes the pools based on the workload. For example, if a benchmark frequently allocates large chunks of contiguous memory, the pool size increases to amortize the cost for each allocation.

4.1.2 Lock Table

As identified in previous work [27, 37], the lock table is a key contention point in DBMSs. Traditionally, the lock table is a centralized data structure (e.g., hash table) keeping track of all the locking information. To lock or unlock an element in the database, the lock table is accessed by latching the whole lock table or part of the table (e.g., one bucket in a hash table). This effectively seriallizes the lock table accesses limiting the system scalability.

In our DBMS, instead of having a centralized lock table or timestamp manager, we implemented these data structures in a per-tuple fashion. Each transaction only latches the tuples that it needs and update the locking or timestamp information. This improves scalability, since all the accesses are decentralized. But per-tuple locking increases the memory overhead because the DBMS now needs to maintain additional meta-data for the lock sharer/waiter information for each tuple. In practice, this meta-data (several bytes) is negligible for large tuples.

4.1.3 Mutexes

Accessing a mutex lock is an expensive operation that requires multiple messages to be sent across the chip. A central critical section protected by a mutex will limit

the scalability of any system (cf. Section 4.3). Therefore, it is important to avoid using mutexes on the critical path. For 2PL, the mutex that protects the centralized deadlock detector is the main bottleneck, while for T/O algorithms it is the mutex used for allocating unique timestamps. In the subsequent sections, we describe the optimizations that we developed to obviate the need for these mutexes.

4.2 Scalable Two-Phase Locking

We next discuss the optimizations for the 2PL algorithms.

4.2.1 Deadlock Detection

For DL_DETECT, the intuitive way to implement the deadlock detector is to use a centralized waits-for graph as a monolithic data structure. The whole graph needs to be locked for detecting deadlocks. However, we found that this deadlock detection algorithm is a bottleneck when multiple threads compete to update their *waits-for* graph. As discussed in Section 4.1.3, having a centralized critical section is inherently non-scalable.

We solved this by partitioning the waits-for graph across cores and making the deadlock detector completely lock-free. Each worker thread has a local queue storing the ID of the transactions it is waiting for. Updating this queue is a local operation (i.e., no cross-core communication), as the thread does not write to the queues of other transactions.

Deadlock detection is also distributed. Detection starts when a worker thread has been waiting for longer than a certain threshold. The detecting thread reads the local queue and find out the IDs of the transactions it is waiting for. If any transaction is no longer active (having already committed or aborted), the transaction ID can be removed from the local queue; otherwise, the queue of its corresponding worker thread is accessed and the transaction IDs are examined. This process continues until a deadlock is found or all the transaction IDs have been examined and no deadlock

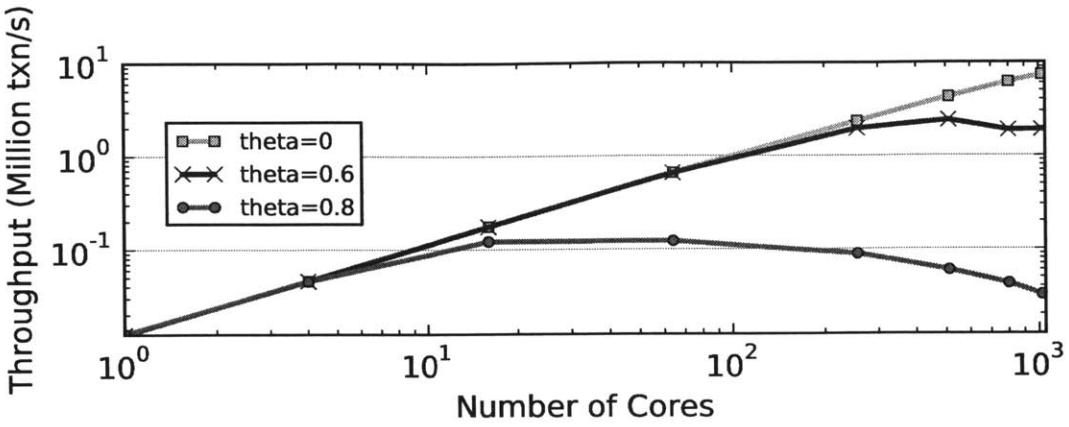


Figure 4-2: **Lock Thrashing** – Results for a write-intensive YCSB workload using the DL_DETECT scheme without deadlock detection. Each transaction acquires locks in their primary key order.

exists.

Note that when the queue of a remote work thread is accessed, it does not need to be locked. This may generate a false negative since the value loaded from the remote queue may not be up-to-date and an existing cycle is not detected; but this algorithm cannot generate a false positive. Furthermore, even if the content of the remote queue is not seen on one pass of deadlock detection, it will certainly be seen on the next pass. And a deadlock can eventually be detected.

4.2.2 Lock Thrashing

Even with improved detection, DL_DETECT still does not scale due to lock thrashing. This occurs when a transaction holds its locks until it commits, blocking all the other concurrent transactions that attempt to acquire those locks. This becomes a problem with high contention and a large number of concurrent transactions, and thus is the main bottleneck of all 2PL schemes.

To demonstrate the impact of thrashing, we executed a write-intensive YCSB workload (i.e., 50/50% read-write mixture) using a variant of DL_DETECT where transactions acquire locks in primary key order. Although this approach is not practical for all workloads, it removes the need for deadlock detection and allows us to

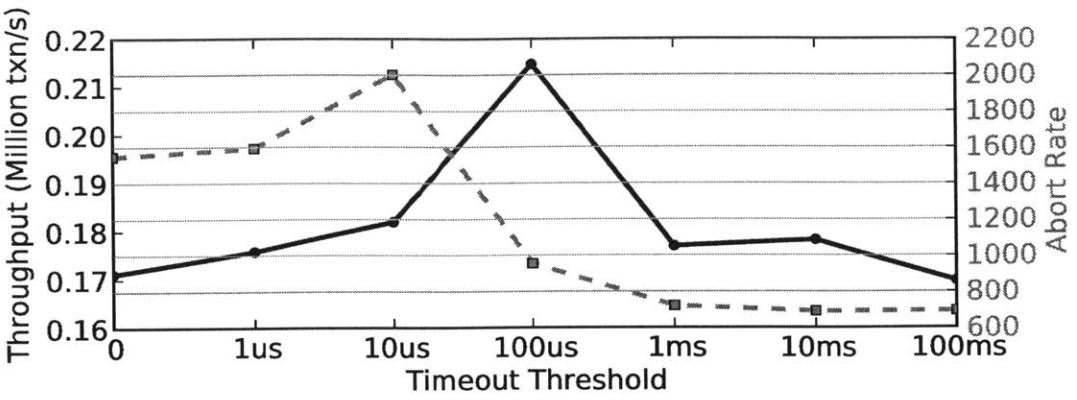


Figure 4-3: **Waiting vs. Aborting** – Results for DL_DETECT with varying timeout threshold running high contention YCSB (*theta*=0.8) at 64 cores.

better observe the effects of thrashing. Fig. 4-2 shows the transaction throughput as a function of the number of cores for different levels of contention. When there is no skew in the workload (*theta*=0), the contention for locks is low and the throughput scales almost linearly. As the contention level increases, however, thrashing starts to occur. With medium contention (*theta*=0.6), the throughput peaks at several hundred cores and then decreases due to thrashing. At the highest contention level (*theta*=0.8), the DBMS's throughput peaks at 16 cores and cannot scale beyond that. Simulation results show that almost all the execution time is spent on waiting for locks. Thus, lock thrashing is the key bottleneck of lock-based approaches that limits scalability in high-contention scenarios.

4.2.3 Waiting vs. Aborting

The thrashing problem can be solved in DL_DETECT by aborting some transactions to reduce the number of active transactions at any point in time. Ideally, this keeps the system running at the highest throughput achieved in Fig. 4-2. We added a *timeout threshold* in the DBMS that causes the system to abort and restart any transaction that has been waiting for a lock for an amount of time greater than the threshold. We note that when timeout is zero, this algorithm is equivalent to NO_WAIT.

We ran the same YCSB workload with high contention using different timeout thresholds on a 64-core CPU. We measure both the throughput and abort rate in the DBMS for the DL_DETECT scheme sweeping the timeout from 0–100 ms.

The results in Fig. 4-3 indicate when the CPU has a small number of cores, it is better to use a shorter timeout threshold. This highlights the trade-off between performance and the transaction abort rate. With a small timeout, the abort rate is high, which reduces the number of running transactions and alleviates the thrashing problem. Using a longer timeout reduces the abort rate at the cost of more thrashing. Therefore, in this paper, we evaluate DL_DETECT with its timeout threshold set to 100μs. In practice, the threshold should be based on an application's workload characteristics.

4.3 Scalable Timestamp Ordering

Finally, we discuss the optimizations that we developed to improve the scalability of the T/O-based algorithms.

4.3.1 Timestamp Allocation:

All T/O-based algorithms make ordering decisions based on transactions' assigned timestamps. The DBMS must therefore guarantee that each timestamp is allocated to only one transaction. A naïve approach to ensure this is to use a *mutex* in the allocator's critical section, but this leads to poor performance. Another common solution is to use an *atomic addition* operation to advance a global logical timestamp (`__sync_fetch_and_add()`). This requires fewer instructions and thus the DBMS's critical section is locked for a smaller period of time than with a mutex. But as we will show, this approach is still insufficient for a 1000-core CPU. We now discuss three timestamp allocation alternatives: (1) atomic addition with batching [43], (2) CPU clocks, and (3) hardware counters.

With the *batched atomic addition* approach, the DBMS uses the same atomic in-

struction to allocate timestamps, but the timestamp manager returns multiple timestamps together in a batch for each request. This method was first proposed in the Silo DBMS [43].

To generate a timestamp using *clock*-based allocation, each worker thread reads a logical clock from its local core and then concatenates it with its thread id. This provides good scalability as long as all the clocks are synchronized. In distributed systems, synchronization is accomplished using software protocols [32] or external clocks [10]. On a many-core CPU, however, this imposes large overhead and thus requires hardware support. As of July 2014, only Intel CPUs support synchronized clocks across cores.

Lastly, the third approach is to use an efficient, built-in *hardware counter*. The counter is physically located at the center of the CPU such that the average distance to each cores is minimized. No existing CPU currently supports this. Thus, we implemented a counter in Graphite where a timestamp request is sent through the on-chip network to increment it atomically in a single cycle.

To determine the maximum rate that the DBMS can allocate timestamps for each method, we ran a micro-benchmark where threads continually acquire new timestamps. The throughput as a function of the number of cores is shown in Fig. 4-4. We first note that *mutex*-based allocation has the lowest performance, with ~1 million timestamps per second (ts/s) on 1024 cores. The *atomic addition* method reaches a maximum of 30 million ts/s with a small number of cores, but throughput decreases with the number of cores down to 8 million ts/s. This is due to the cache coherence traffic from writing back and invalidating the last copy of the corresponding cache line for every timestamp. This takes one round trip of communication across the chip or ~100 cycles for a 1024-core CPU, which translates to a maximum throughput of 10 million ts/s at 1GHz frequency. Batching these allocations does help, but it causes performance issues when there is contention (see below). The hardware-based solutions are able to scale with the number of cores. Because incrementing the timestamp takes only one cycle with the *hardware counter*-based approach, this method achieves a maximum throughput of 1 billion ts/s. The performance gain comes from

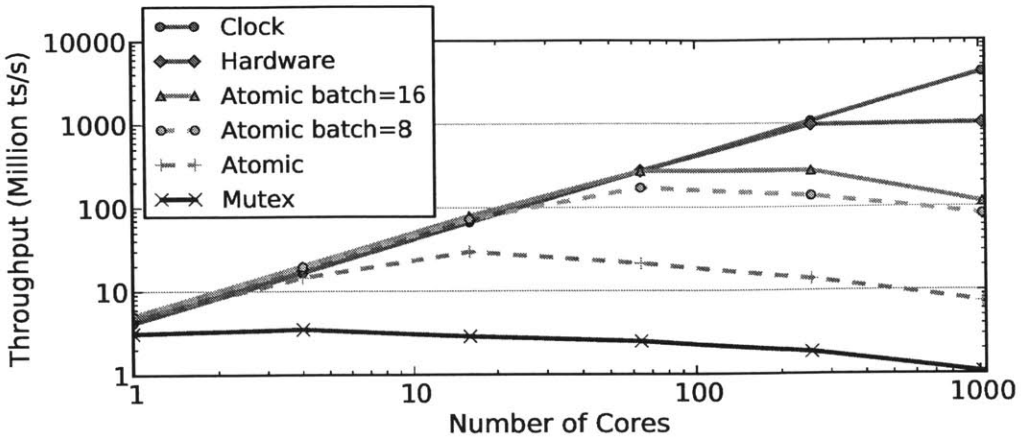
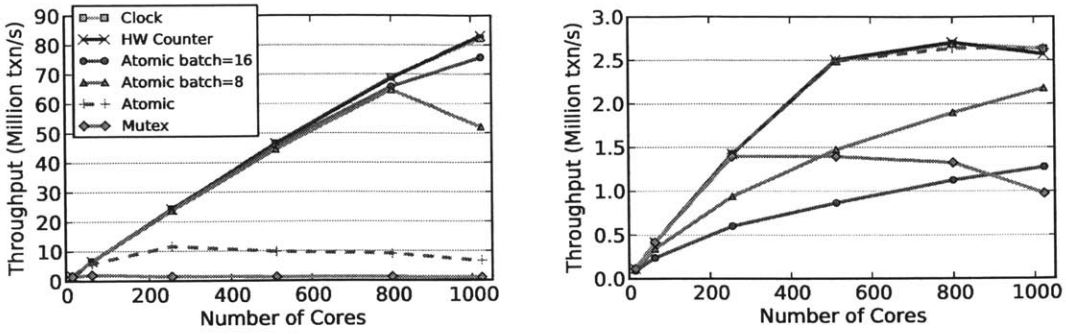


Figure 4-4: **Timestamp Allocation Micro-benchmark** – Throughput measurements for different timestamp allocation methods.

removing the coherence traffic by executing the addition operation remotely. The *clock*-based approach has ideal (i.e., linear) scaling, since this solution is completely decentralized.

We also tested the different allocation schemes in the DBMS to see how they perform for real workloads. For this experiment, we executed a write-intensive YCSB workload with two different contention levels using the TIMESTAMP scheme. The results in Fig. 4-5a show that with no contention, the relative performance of the allocation methods are the same as in Fig. 4-4. When there is contention, however, the trends in Fig. 4-5b are much different. First, the DBMS's throughput with the *batched atomic addition* method is much worse. This is because when a transaction is restarted due to a conflict, it gets restarted in the same worker thread and is assigned the next timestamp in the last batch. But this new timestamp will also be less than the one for the other transaction that caused the abort, and thus it will continually restart until the thread fetches a new batch. The non-batched *atomic addition* method performs as well as the *clock* and *hardware counter* approaches. Hence, for this paper the DBMS uses *atomic addition* without batching to allocate timestamps because the other approaches require specialized hardware support that is currently not available on all CPUs.



(a) No Contention

(b) Medium Contention

Figure 4-5: **Timestamp Allocation** – Throughput of the YCSB workload using TIMESTAMP with different timestamp allocation methods.

4.3.2 Distributed Validation

The original OCC algorithm contains a critical section at the end of the read phase, where the transaction's read set is compared to previous transactions' write sets to detect conflicts. Although this step is short, as mentioned above, any `mutex`-protected central critical section severely hurts scalability. We solve this problem by using per-tuple validation that breaks up this check into smaller operations. This is similar to the approach used in Hekaton [30] but it is simpler, since we only support a single version of each tuple.

Specifically, our OCC works as follows. Each data element has a write timestamp ($D.wts$) indicating the timestamp of its last update. And each transaction has two timestamps, the start timestamp ($T.sts$) and end timestamp ($T.ets$) indicating when the transaction starts and finishes respectively. The transaction also keeps track of its read set (*read_set*) and write set (*write_set*).

Transaction starts

$T.sts = get\_global\_timestamp()$

Tuple read/write

if $T.sts < D.wts$

Abort

else

Add the operation to the read/write set

Validation

Lock all the rows in the read_set and write_set following the primary key order

T.ets = get_global_timestamp()

For each read

if T.sts < D.wts Abort

For each write

if T.ets < D.wts Abort

Write Phase

For each write

D.wts = T.ets

update the element in the database

release all locks and commit

The algorithm described above is more scalable since only the tuples being accessed need to be locked for validation. Transactions touching different data elements can thus validate in parallel with no contention.

4.3.3 Local Partitions

We optimized the original H-STORE protocol to take advantage of shared memory. Because the DBMS's worker threads run in a single process, we allow multi-partition transactions to access tuples at remote partitions directly instead of sending query requests that are executed by the remote partitions' worker threads. This allows for a simpler implementation that is faster than using intra-process communication. With this approach, the data is not physically partitioned since on-chip communication latency is low. Read-only tables are accessed by all threads without replication, thus reducing the memory footprint. Finally, we use the same timestamp allocation optimizations from above to avoid the mandatory wait time to account for clock skew [39].

Chapter 5

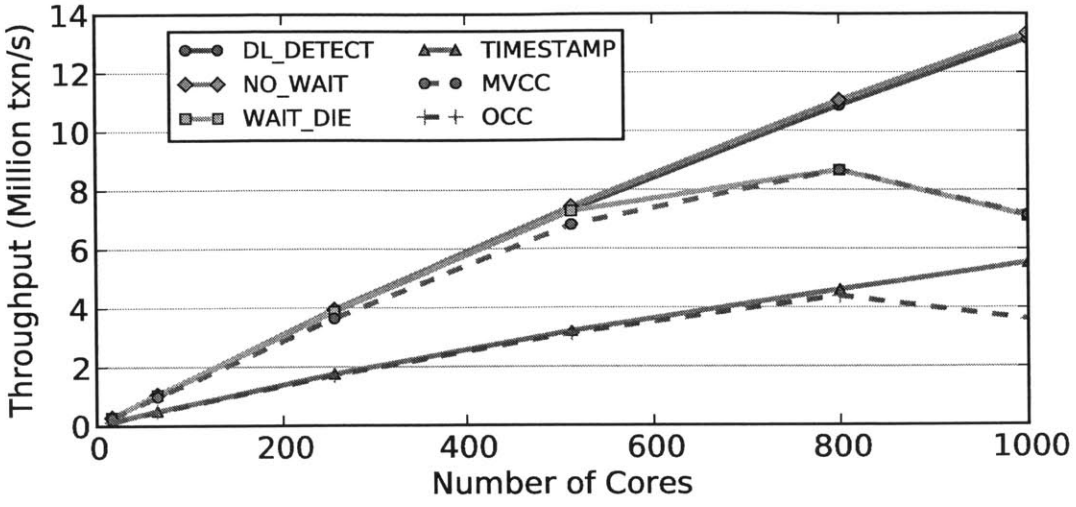
Experimental Analysis

We now present the results from our analysis of the different concurrency control schemes. Our experiments are grouped into two categories: (1) *scalability* and (2) *sensitivity* evaluations. For the former, we want to determine how well the schemes perform as we increase the number of cores. We scale the number of cores up to 1024 while fixing the workload parameters. With the sensitivity experiments, we vary a single workload parameter (e.g., transaction access skew). We report the DBMS's total simulated throughput as well as a breakdown of the amount of time that each worker thread spends in the different parts of the system listed in Section 3.2.

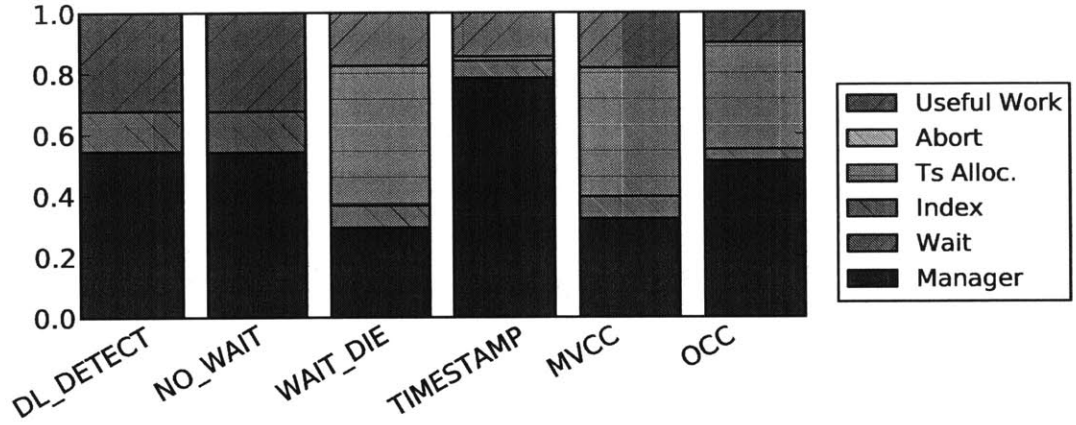
We begin with an extensive analysis of the YCSB workload. The nature of this workload allows us to change its parameters and create a variety of scenarios that stress the concurrency control schemes in different ways. Next, we analyze the TPC-C workload, where we vary the number of warehouses and observe the impact on the throughput of the algorithms. The H-STORE scheme is excluded from our initial experiments and is only introduced in Section 5.5 when we analyze database partitioning.

5.1 Read-Only Workload

In this first scalability analysis experiment, we executed a YCSB workload comprising read-only transactions with a uniform access distribution (i.e., theta=0). Each trans-



(a) Total Throughput

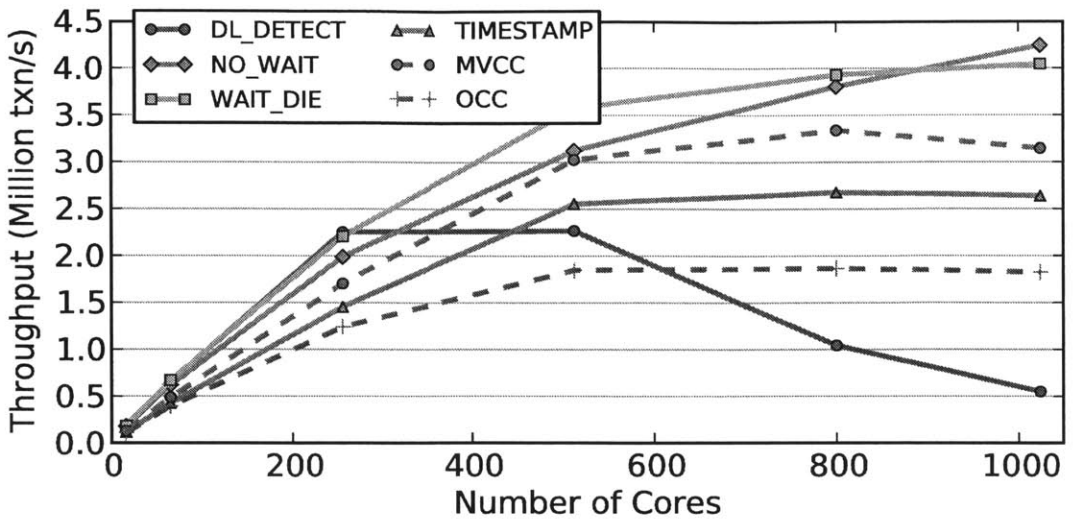


(b) Runtime Breakdown (1024 cores)

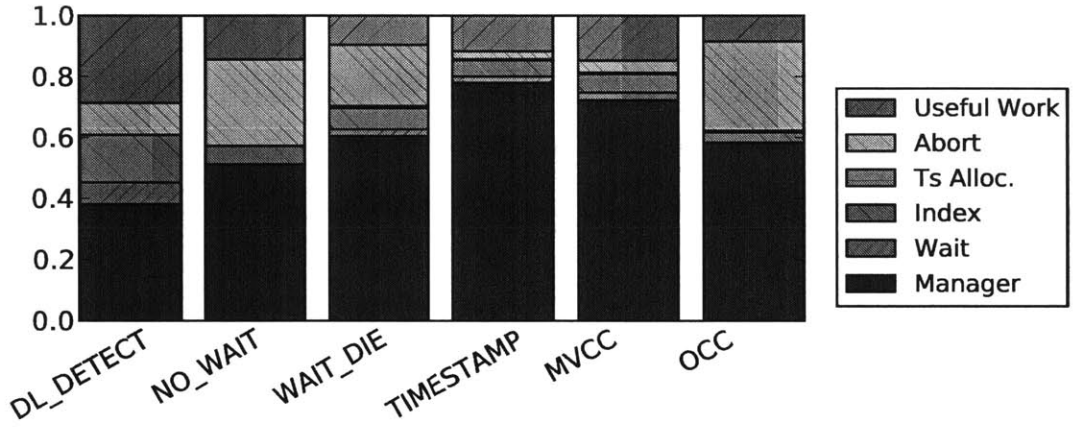
Figure 5-1: **Read-only Workload** – Results for a read-only YCSB workload.

action executes 16 separate tuple reads at a time. This provides a baseline for each concurrency control scheme before we explore more complex workload arrangements.

In a perfectly scalable DBMS, the throughput should increase linearly with the number of cores. This is not the case, however, for the T/O schemes in Fig. 5-1a. The time breakdown in Fig. 5-1b indicates that timestamp allocation becomes the bottleneck with a large core count. OCC hits the bottleneck even earlier since it needs to allocate timestamps twice per transaction (i.e., at transaction start and before the validation phase). Both OCC and TIMESTAMP have significantly worse performance



(a) Total Throughput



(b) Runtime Breakdown (512 cores)

Figure 5-2: **Write-Intensive Workload (Medium Contention)** – Results for YCSB workload with medium contention (*theta*=0.6).

than the other algorithms regardless of the number of cores. These algorithms waste cycles because they copy tuples to perform a read, whereas the other algorithms read tuples in place.

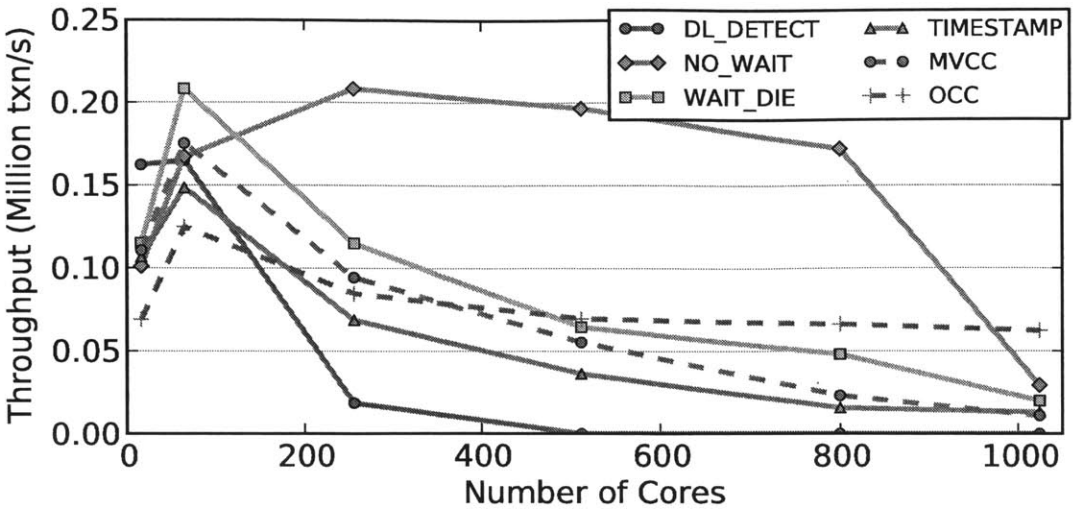
5.2 Write-Intensive Workload

A read-only workload represents an optimistic (and unrealistic) scenario, as it generates no data contention. But even if we introduce writes in the workload, the large size of the dataset means that the probability that any two transactions access the same tuples at the same time is small. In reality, the access distribution of an OLTP application is rarely uniform. Instead, it tends to follow a Zipfian skew, where certain tuples are more likely to be accessed than others. This can be from either skew in the popularity of elements in the database or skew based on temporal locality (i.e., newer tuples are accessed more frequently). As a result, this increases contention because transactions compete to access the same data.

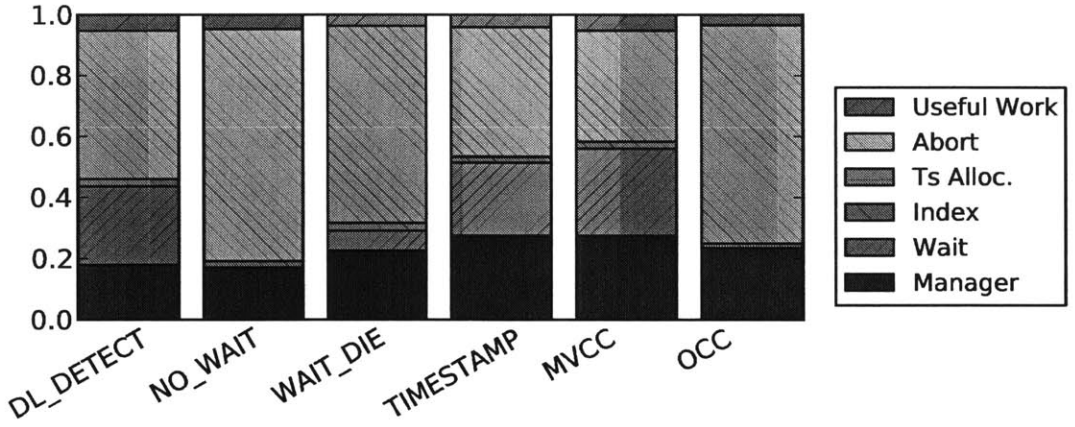
We executed a write-intensive YCSB workload comprising transactions that access 16 tuples at time. Within each transaction, each of these accesses will modify the tuple with a 50% probability. The amount of skew in the workload is determined by the parameter *theta* (cf. Section 3.3). We use the medium and high contention levels for the transactions' access patterns.

The medium contention results in Fig. 5-2 show that NO_WAIT and WAIT_DIE are the only 2PL schemes that scales past 512 cores. NO_WAIT scales better than WAIT_DIE. For DL_DETECT, the breakdown in Fig. 5-2b indicates that the DBMS spends a larger percentage of its time waiting in these schemes. DL_DETECT is inhibited by lock thrashing at 256 cores. WAIT_DIE is more conservative, but it still suffers from lock thrashing at larger core counts. NO_WAIT is the most scalable because it eliminates this waiting. We note, however, that both NO_WAIT and WAIT_DIE have a high transaction abort rate. This is not an issue in our experiments because restarting an aborted transaction has low overhead; the time it takes to undo a transaction is slightly less than the time it takes to re-execute the transactions queries. But in reality, the overhead may be larger for workloads where transactions have to rollback changes to multiple tables, indexes, and materialized views.

The results in Fig. 5-2a also show that the T/O algorithms perform well in general. Both TIMESTAMP and MVCC are able to overlap operations and reduce the waiting



(a) Total Throughput



(b) Runtime Breakdown (64 cores)

Figure 5-3: **Write-Intensive Workload (High Contention)** – Results for YCSB workload with high contention (*theta*=0.8).

time. MVCC performs slightly better since it keeps multiple versions of a tuple and thus can serve read requests even if they have older timestamps. OCC does not perform as well because it spends a large portion of its time aborting transactions; the overhead is worse since each transaction has to finish before the conflict is resolved.

With higher contention, the results in Fig. 5-3 show that performance of all of the algorithms is much worse. Fig. 5-3a shows that almost all of the schemes are unable to scale to more than 64 cores. Beyond this point, the DBMS's throughput stops

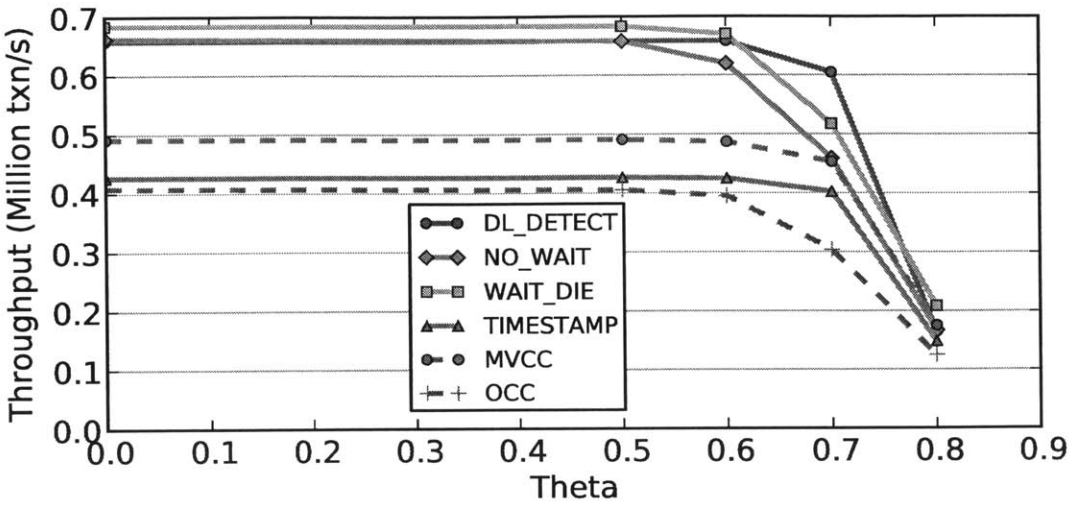


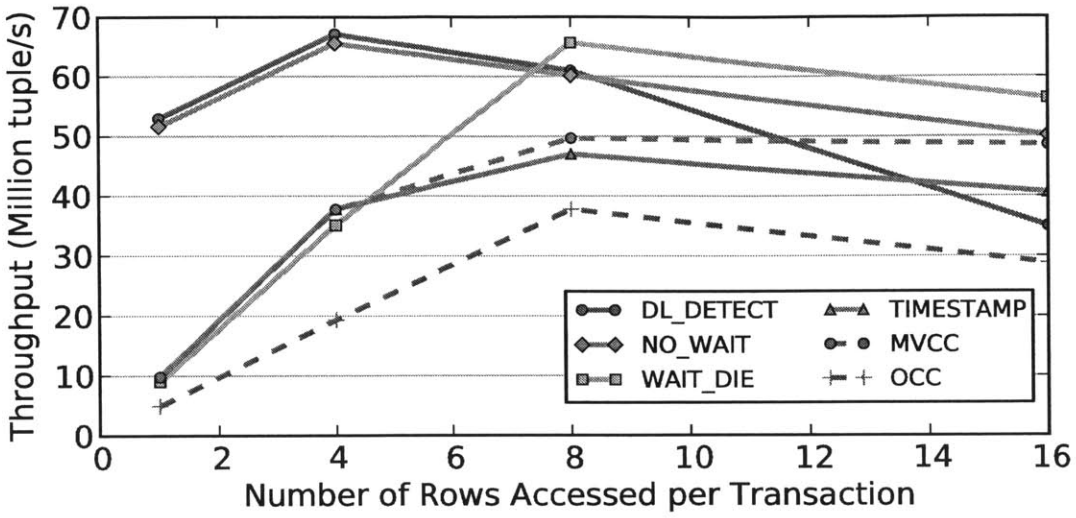
Figure 5-4: **Write-Intensive Workload (Variable Contention)** – Results for YCSB workload with varying level of contention on 64 cores.

increasing and there is no performance benefit to the increased core count. NO_WAIT initially outperforms all the others, but then succumbs to lock thrashing (cf. Fig. 4-2). Surprisingly, OCC performs the best on 1024 cores. This is because although a large number of transactions conflict and have to abort during the validation phase, one transaction is always allowed to commit. The time breakdown in Fig. 5-3b shows that the DBMS spends a larger amount of time aborting transactions in every scheme.

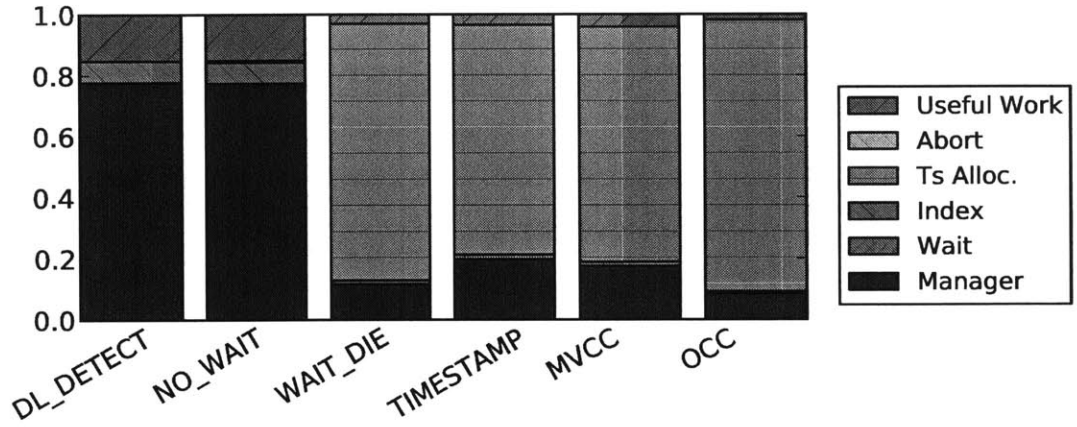
To better understand when each scheme begins to falter with increased contention, we fixed the number of cores to 64 and performed a sensitivity analysis on the skew parameter (*theta*). The results in Fig. 5-4 indicate that for *theta* values less than 0.6, the contention has little effect on the performance. But for higher settings, there is a sudden drop in throughput that renders all algorithms non-scalable and approaches zero for values greater than 0.8.

5.3 Working Set Size

The number of tuples accessed by a transaction is another factor that impacts scalability. When a transaction's working set is large, it increases the likelihood that the same data is accessed by concurrent transactions. For 2PL algorithms, this in-



(a) Total Throughput



(b) Runtime Breakdown (transaction length = 1)

Figure 5-5: **Working Set Size** – The number of tuples accessed per core on 512 cores for transactions with a varying number of queries (*theta*=0.6).

creases the length of time that locks are held by a transaction. With T/O, however, longer transactions may reduce timestamp allocation contention. In this experiment, we vary the number of tuples accessed per transaction in a write-intensive YCSB workload. Because short transactions leads to higher throughput, we measure the number of tuples accessed per second, rather than transactions completed. We use the medium skew setting (*theta*=0.6) and fix the core count to 512.

The results in Fig. 5-5 show that when transactions are short, the lock contention is

low. DL_DETECT and NO_WAIT have the best performance in this scenario, since there are few deadlocks and the number of aborts is low. But as the transactions' working set size increases, the performance of DL_DETECT degrades due to the overhead of thrashing. For the T/O algorithms and WAIT_DIE, the throughput is low when the transactions are short because the DBMS spends a majority of its time allocating timestamps. But as the transactions become longer, the timestamp allocation cost is amortized. OCC performs the worst because it allocates double the number of timestamps as the other schemes for each transaction.

Fig. 5-5b shows the time breakdown for transaction length equals to one. Again, we see that the T/O schemes spend most of their execution time allocating timestamps. As the transactions become longer, Figs. 5-1b and 5-2b shows that the allocation is no longer the main bottleneck. The results in Fig. 5-5 also show that the T/O-based algorithms are more tolerant to contention than DL_DETECT.

5.4 Read/Write Mixture

Another important factor for concurrency control is the read/write mixtures of transactions. More writes leads to more contention that affect the algorithms in different ways. For this experiment, we use YCSB on a 64 core configuration and vary the percentage of read queries executed by each transaction. Each transaction executes 16 queries using the high skew setting (*theta*=0.8).

The results in Fig. 5-6 indicate that all of the algorithms achieve better throughput when there are more read transactions. At 100% reads, the results match the previous read-only results in Fig. 5-1a. TIMESTAMP and OCC do not perform well because they copy tuples for reading. MVCC stand out as having the best performance when there are small number of write transactions. This is also an example of where supporting non-blocking reads through multiple versions is most effective; read queries access the correct version of a tuple based on timestamps and do not need to wait for a writing transaction. This is a key difference from TIMESTAMP, where late arriving queries are rejected and their transactions are aborted.

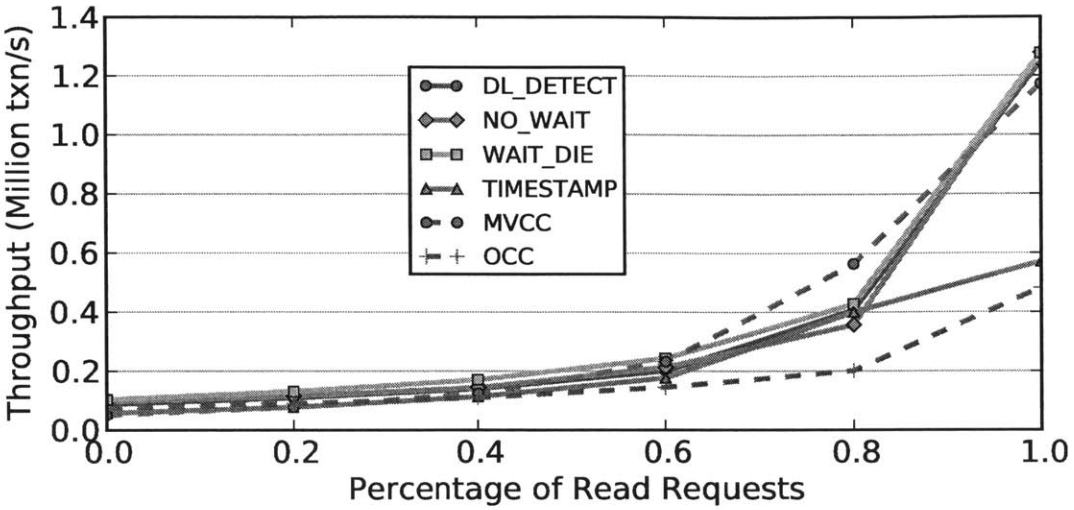


Figure 5-6: **Read/Write Mixture** – Results for YCSB with a varying percentage of read-only transactions with high contention (*theta*=0.8).

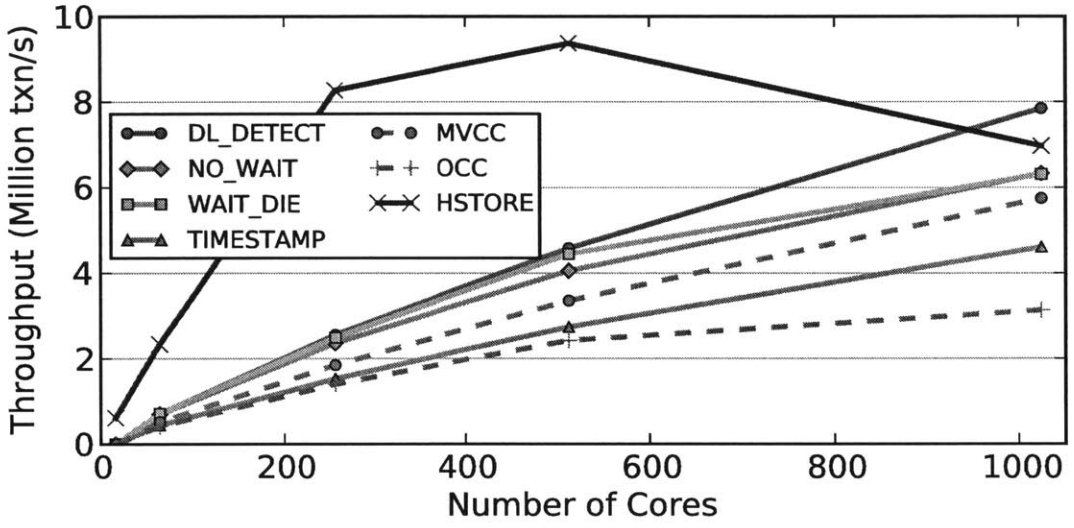


Figure 5-7: **Database Partitioning** – Results for a read-only workload on a partitioned YCSB database. The transactions access the database based on a uniform distribution (*theta*=0.0).

5.5 Database Partitioning

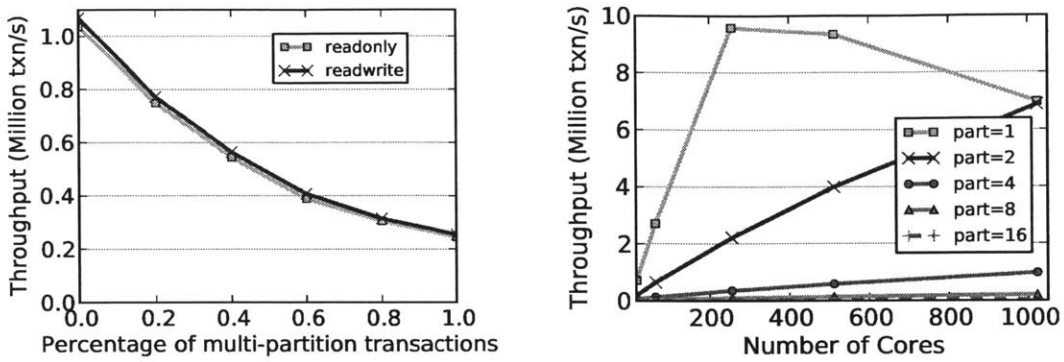
Up to this point in our analysis, we assumed that the database is stored as a single partition in memory and that all worker threads can access any tuple. With the H-STORE scheme, however, the DBMS splits the database into disjoint subsets to in-

crease scalability [39]. This approach achieves good performance only if the database is partitioned in such a way that enables a majority of transactions to only need to access data at a single partition [35]. H-STORE does not work well when the workload contains multi-partition transactions because of its coarse-grained locking scheme. It also matters how many partitions each transaction accesses; for example, H-STORE will still perform poorly even with a small number of multi-partition transactions if they access all partitions. To explore these issues in a many-core setting, we first compare H-STORE to the six other schemes under ideal conditions. We then analyze its performance with multi-partition transactions.

We divide the YCSB database into the same number of partitions as the number of cores in each trial. Since YCSB only has one table, we use a simple hashing strategy to assign tuples to partitions based on their primary keys so that each partition stores approximately the same number of records. These tests use a write-intensive workload where each transaction executes 16 queries that all use index look-ups without any skew (*theta*=0.0). We also assume that the DBMS knows what partition to assign each transaction to at runtime before it starts [35].

In the first experiment, we executed a workload comprised only of single-partition transactions. The results in Fig. 5-7 show that H-STORE outperforms all other schemes up to 800 cores. Since it is especially designed to take advantage of partitioning, it has a much lower overhead for locking than the other schemes. But because H-STORE also depends on timestamp allocation for scheduling, it suffers from the same bottleneck as the other T/O-based schemes. As a result, the performance degrades at higher core counts. For the other schemes, partitioning does not have a significant impact on throughput. It would be possible, however, to adapt their implementation to take advantage of partitioning [37].

We next modified the YCSB driver to vary the percentage of multi-partition transactions in the workload and deployed the DBMS on a 64-core CPU. The results in Fig. 5-8a illustrate two important aspects of the H-STORE scheme. First, there is no difference in performance whether or not the workload contains transactions that modify the database; this is because of H-STORE's locking scheme. Second,



(a) Multi-Partition Percentage

(b) Partitions per Transaction

Figure 5-8: **Multi-Partition Transactions** – Sensitivity analysis of the H-STORE scheme for YCSB workloads with multi-partition transactions.

the DBMS's throughput degrades as the number of multi-partition transactions in the workload increases because they reduce the amount of parallelism in the system [35, 43].

Lastly, we executed YCSB with 10% multi-partition transactions and varied the number of partitions that they access. The DBMS's throughput for the single-partition workload in Fig. 5-8b exhibits the same degradation due to timestamp allocation as H-STORE in Fig. 5-7. This is also why the throughputs for the one- and two-partition workloads converge at 1000 cores. The DBMS does not scale with transactions accessing four or more partitions because of the reduced parallelism and increased cross-core communication.

5.6 TPC-C

Finally, we analyze the performance of all the concurrency control algorithms when running the TPC-C benchmark. The transactions in TPC-C are more complex than those in YCSB and is representative of a large class of OLTP applications. For example, they access multiple tables with a *read-modify-write* access pattern and the output of some queries are used as the input for subsequent queries in the same transaction. TPC-C transactions can also abort because of certain conditions in their

program logic, as opposed to only because the DBMS detected a conflict.

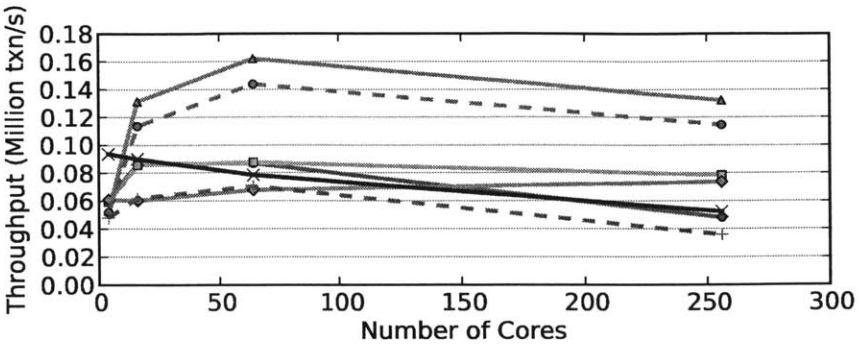
The workload in each trial comprises 50% `NewOrder` and 50% `Payment` transactions. These two make up 88% of the default TPC-C mix and are the most interesting in terms of complexity. Supporting the other transactions would require additional DBMS features, such as B-tree latching for concurrent updates. This would add additional overhead to the system, and thus we defer the problem of scaling indexes for many-core CPUs as future work.

The size of TPC-C databases are typically measured by the number of warehouses. The warehouse is the root entity for almost all tables in the database. We follow the TPC-C specification where ~10% of the `NewOrder` transactions and ~15% of the `Payment` transactions access a "remote" warehouse. For partitioned-based schemes, such as H-STORE, each partition consists of all the data for a single warehouse [39]. This means that the remote warehouse transactions will access multiple partitions.

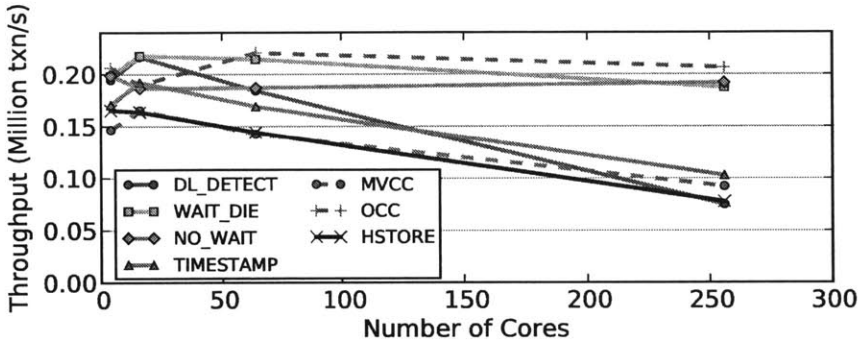
We performed our evaluation using two different database sizes. We first execute the TPC-C workload on a 4-warehouse database with 100MB of data per warehouse (0.4GB in total). This allows us to evaluate the algorithms when there are more worker threads than warehouses. We then execute the same workload again on a 1024-warehouse database. Due to memory constraints of running in the Graphite simulator, we reduced the size of this database to 26MB of data per warehouse (26GB in total). This does not affect our measurements because the number of tuples accessed by each transaction is independent of the database size.

5.6.1 4 Warehouses

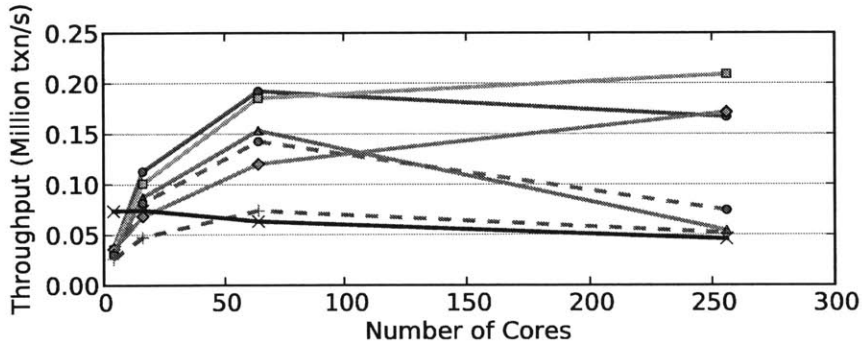
The results in Fig. 5-9 show that all of the schemes fail to scale for TPC-C when there are fewer warehouses than cores. With H-STORE, the DBMS is unable to utilize extra cores because of its partitioning scheme; the additional worker threads are essentially idle. For the other schemes, the results in Fig. 5-9c show that they are able to scale up to 64 cores for the `NewOrder` transaction. TIMESTAMP, MVCC, and OCC have worse scalability due to high abort rates. DL_DETECT does not scale due to thrashing and deadlocks. But the results in Fig. 5-9b show that no scheme scales for the `Payment`



(a) `Payment` + `NewOrder`



(b) `Payment` only



(c) `NewOrder` only

Figure 5-9: **TPC-C (4 warehouses)** – Results for the TPC-C workload running up to 256 cores.

transaction. The reason for this is that every `Payment` transaction updates a single field in the warehouse (`W_YTD`). This means that either the transaction (1) must acquire an exclusive lock on the corresponding tuple (i.e., DL_DETECT) or (2) issue a pre-write on that field (i.e., T/O-based algorithms). If the number of threads is greater than

the number of warehouses, then updating the warehouse table becomes a bottleneck.

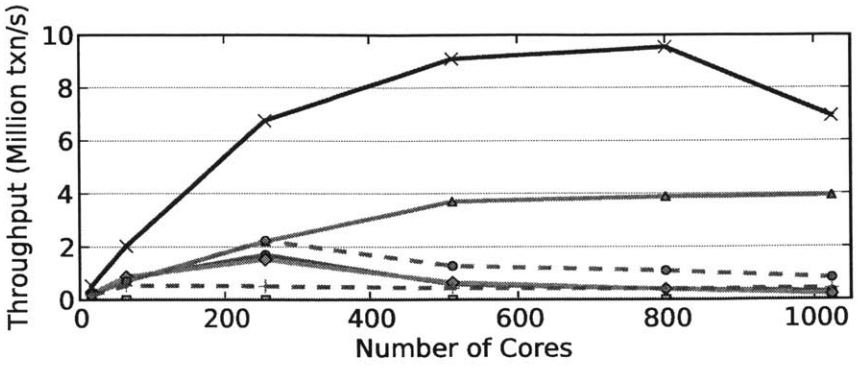
In general, the main problem for these two transactions is the contention on updating the `WAREHOUSE` table. Each `Payment` transaction updates its corresponding warehouse entry and each `NewOrder` will read it. For the 2PL-based algorithms, these read and write operations block each other. Two of the T/O-based algorithms, TIMESTAMP and MVCC, outperform the other schemes because their write operations are not blocked by reads. This eliminates the lock blocking problem in 2PL. As a result, the `NewOrder` transactions can execute in parallel with `Payment` transactions.

5.6.2 1024 Warehouses

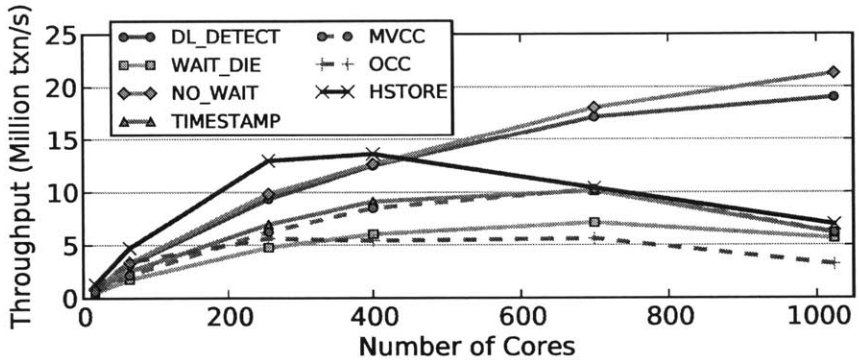
We next execute the TPC-C workload with 1024 warehouses with up to 1024 cores. Once again, we see in Fig. 5-10 that no scheme is able to scale. The results indicate that unlike in Section 5.6.1, the DBMS's throughput is limited by `NewOrder` transactions. This is due to different reasons for each scheme.

With almost all the schemes, the main bottleneck is the overhead of maintaining locks and latches, which occurs even if there is no contention. For example, the `NewOrder` transaction reads tuples from the read-only `ITEM` table, which means for the 2PL schemes that each access creates a shared-lock entry in the DBMS. With a large number of concurrent transactions, the lock meta-data becomes large and thus it takes longer to update them. OCC does not use such locks while a transaction runs, but it does use latches for each tuple accessed during the validation phase. Acquiring these latches becomes an issue for transactions with large footprints, like `NewOrder`. Although MVCC also does not have locks, each read request generates a new history record, which increases memory traffic. We note, however, that all of this is technically unnecessary work because the `ITEM` table is never modified.

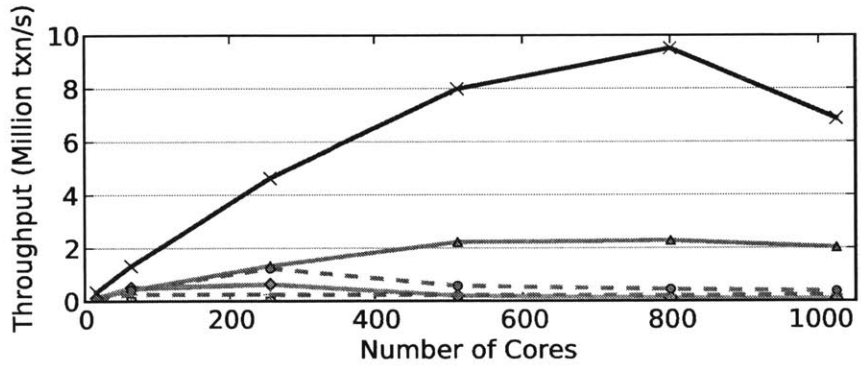
The results in Fig. 5-10b indicate that when the number of warehouses is the same or greater than the number of worker threads, the bottleneck in the `Payment` transaction is eliminated. This improves the performance of all schemes. For T/O schemes, however, the throughput becomes too high at larger core counts and thus they are inhibited by timestamp allocation. As a result, they are unable to achieve



(a) `Payment` + `NewOrder`



(b) `Payment` only



(c) `NewOrder` only

Figure 5-10: **TPC-C (1024 warehouses)** – Results for the TPC-C workload running up to 1024 cores.

higher than ~10 million txn/s. This is the same scenario as Fig. 5-5a where 2PL outperforms T/O for short transactions.

H-STORE performs the best overall due to its ability to exploit partitioning even

with ~12% multi-partition transactions in the workload. This corroborates results from previous studies that show that H-STORE outperforms other approaches when less than 20% workload comprises multi-partition transactions [35, 43]. At 1024 cores, however, it is limited by the DBMS's timestamp allocation.

Chapter 6

Discussion

We now discuss the results of the previous sections and propose solutions to avoid these scalability issues for many-core DBMSs.

6.1 DBMS Bottlenecks

Our evaluation shows that all seven concurrency control schemes fail to scale to a large number of cores, but for different reasons and conditions. Table 6.1 summarizes the findings for each of the schemes. In particular, we identified several bottlenecks to scalability: (1) lock thrashing, (2) preemptive aborts, (3) deadlocks, (4) timestamp allocation, and (5) memory-to-memory copying.

Thrashing happens in any waiting-based algorithm. As discussed in Section 4.2, thrashing is alleviated by proactively aborting. This leads to the trade-off between aborts and performance. In general, the results in Section 5.2 showed that for high-contention workloads, a non-waiting deadlock prevention scheme (NO_WAIT) performs much better than deadlock detection (DL_DETECT).

Although no single concurrency control scheme performed the best for all workloads, one may outperform the others under certain conditions. Thus, it may be possible to combine two or more classes of algorithms into a single DBMS and switch between them based on the workload. For example, a DBMS could use DL_DETECT for workloads with little contention, but switch to NO_WAIT or a T/O-based algo-

rithm when transactions are taking too long to finish due to thrashing. One could also employ a hybrid approach, such as MySQL's DL_DETECT + MVCC scheme, where read-only transactions use multi-versioning and all others use 2PL.

These results also make it clear that new hardware support is needed to overcome some of these bottlenecks. For example, all of the T/O schemes suffer from the timestamp allocation bottleneck when the throughput is high. Using the *atomic addition* method when the core count is large causes the worker threads to send many messages across the chip to modify the timestamp. We showed in Section 4.3 how the *clock* and *hardware counter* methods performed the best without the drawbacks of batching. Thus, we believe that they should be included in future CPU architectures.

We also saw that memory issues cause slowdown in some of the schemes. One way to alleviate this problem is to add a hardware accelerator on the CPU to do memory copying in the background. This would eliminate the need to load all data through the CPU's pipeline. We also showed in Section 4.1 how `malloc` was another bottleneck and that we were able to overcome it by developing our own implementation that supports dynamic pool resizing. But with a large number of cores, these pools become too unwieldy to manage in a global memory controller. We believe that future CPUs will need to switch to decentralized or hierarchical memory controllers to provide faster memory allocation.

6.2 Multi-core vs. Multi-node Systems

Distributed DBMSs are touted for being able to scale beyond what a single-node DBMS can support [39]. This is especially true when the number of CPU cores and the amount of memory available on a node is small. But moving to a multi-node architecture introduces a new performance bottleneck: *distributed transactions* [4]. Since these transactions access data that may not be on the same node, the DBMS must use an atomic commit protocol, such as *two-phase commit* [17]. The coordination overhead of such protocols inhibits the scalability of distributed DBMSs because the communication between nodes over the network is slow. In contrast, communication

2PL	**DL_DETECT**	Scales under low-contention. Suffers from lock thrashing.
	NO_WAIT	Has no centralized point of contention. Highly scalable. Very high abort rate.
	WAIT_DIE	Suffers from lock thrashing and timestamp bottleneck.
T/O	**TIMESTAMP**	High overhead from copying data locally. Non-blocking writes. Suffers from timestamp bottleneck.
	MVCC	Performs well w/ read-intensive workload. Non-blocking reads and writes. Suffers from timestamp bottleneck.
	OCC	High overhead for copying data locally. High abort cost. Suffers from timestamp bottleneck.
	H-STORE	The best algorithm for partitioned workloads. Suffers from multi-partition transactions and timestamp bottleneck.

Table 6.1: A summary of the bottlenecks for each concurrency control scheme evaluated in Chapter 5.

between threads in a shared-memory environment is much faster. This means that a single many-core node with a large amount of DRAM might outperform a distributed DBMS for all but the largest OLTP applications [43].

It may be that for multi-node DBMSs two levels of abstraction are required: a shared-nothing implementation between nodes and a multi-threaded shared-memory DBMS within a single chip. This hierarchy is common in high-performance computing applications. More work is therefore needed to study the viability and challenges of such hierarchical concurrency control in an OLTP DBMS.

Chapter 7

Related Work

The work in [40] is one of the first hardware analysis of a DBMS running an OLTP workload. Their evaluation focused on multi-processor systems, such as how to assign processes to processors to avoid bandwidth bottlenecks. Another study [38] measured CPU stall times due to cache misses in OLTP workloads. This work was later expanded in [3] and more recently by [42, 36].

With the exception of H-STORE [15, 23, 39, 44] and OCC [29], all other concurrency control schemes implemented in our test-bed are derived from the seminal surveys by Bernstein et al. [4, 6]. In recent years, there have been several efforts towards improving the shortcomings of these classical implementations [12, 25, 33, 43]. Other work includes standalone lock managers that are designed to be more scalable on multi-core CPUs [37, 27]. We now describe these systems in further detail and discuss why they are still unlikely to scale on future many-core architectures.

Shore-MT [25] is a multi-threaded version of Shore [8] that employs a deadlock detection scheme similar to DL_DETECT. Much of the improvements in Shore-MT come from optimizing bottlenecks in the system other than concurrency control, such as logging [26]. The system still suffers from the same thrashing bottleneck as DL_DETECT on high contention workloads.

DORA is an OLTP execution engine built on Shore-MT [33]. Instead of assigning transactions to threads, as in a traditional DBMS architecture, DORA assigns threads to partitions. When a transaction needs to access data at a specific partition, its

handle is sent to the corresponding thread for that partition where it then waits in a queue for its turn. This is similar to H-STORE's partitioning model, except that DORA supports multiple record-level locks per partition (instead of one lock per partition) [34]. We investigated implementing DORA in our DBMS but found that it could not be easily adapted and requires a separate system implementation.

The authors of Silo [43] also observed that global critical sections are the main bottlenecks in OCC. To overcome this, they use a decentralized validation phase based on *batched atomic addition* timestamps. But as we showed in Section 4.3, the DBMS must use large batches when deployed on a large number of cores to amortize the cost of centralized allocation. This batching in turn increases the system's latency under contention.

Hekaton [12] is a main memory table extension for Microsoft's SQL Server that uses a variant of MVCC with lock-free data structures [30]. The administrator designates certain tables as in-memory tables that are then accessed together with regular, disk-resident tables. The main limitation of Hekaton is that timestamp allocation suffers from the same bottleneck as the other T/O-based algorithms evaluated in this paper.

The VLL centralized lock manager uses per-tuple 2PL to remove contention bottlenecks [37]. It is an optimized version of DL_DETECT that requires much smaller storage and computation overhead than our implementation when the contention is low. VLL achieves this by partitioning the database into disjoint subsets. Like H-STORE, this technique only works when the workload is partitionable. Internally, each partition still has a critical section that will limit scalability at high contention workloads.

The work in [27] identified latch contention as the main scalability bottleneck in MySQL. They removed this contention by replacing the *atomic-write-after-read* synchronization pattern with a *read-after-write* scheme. They also proposed to pre-allocate and deallocate locks in bulk to improve scalability. This system, however, is still based on centralized deadlock detection and thus will perform poorly when there is contention in the database. In addition, their implementation requires the usage

of global barriers that will be problematic at higher core counts.

Others have looked into using the software-hardware co-design approach for improving DBMS performance. The "bionic database" project [24] is similar to our proposal, but it focuses on implementing OLTP DBMS operations in FPGAs instead of new hardware directly on the CPU. Other work is focused on OLAP DBMSs and thus is not applicable to our problem domain. For example, an FPGA-based SQL accelerator proposed in [11] filters in-flight data moving from a data source to a data sink. It targets OLAP applications by using the FPGA to accelerate the projection and restriction operations. The Q100 project is a special hardware co-processor for OLAP queries [45]. It assumes a column-oriented database storage and provides special hardware modules for each SQL operator.

Chapter 8

Conclusion

This thesis studied the scalability bottlenecks in concurrency control algorithms for many-core CPUs. We implemented a lightweight main memory DBMS with a pluggable architecture that supports seven concurrency control schemes. We ran our DBMS in a distributed CPU simulator that provides a virtual environment of 1000 cores. Our results show that none of the algorithms are able to get good performance at such a high core count in all situations. For lower core configurations, we found that 2PL-based schemes are good at handling short transactions with low contention that are common in key-value workloads. Whereas T/O-based algorithms are good at handling higher contention with longer transactions that are more common in complex OLTP workloads. Although it may seem like all hope is lost, we proposed several research directions that we plan to explore to rectify these scaling issues.

Bibliography

[1] VoltDB. `http://voltdb.com`.

[2] Intel brings supercomputing horsepower to big data analytics. `http://intel.ly/18A03EM`, November 2013.

[3] A. Ailamaki, D. J. DeWitt, M. D. Hill, and D. A. Wood. DBMSs on a modern processor: Where does time go? In *VLDB*, pages 266–277, 1999.

[4] P. A. Bernstein and N. Goodman. Concurrency control in distributed database systems. *ACM Comput. Surv.*, 13(2):185–221, 1981.

[5] P. A. Bernstein and N. Goodman. Multiversion concurrency control - theory and algorithms. *ACM Trans. Database Syst.*, 8(4):465–483, Dec. 1983.

[6] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*, chapter 5. 1987.

[7] P. A. Bernstein, D. Shipman, and W. Wong. Formal aspects of serializability in database concurrency control. *IEEE Transactions on Software Engineering*, 5(3):203–216, 1979.

[8] M. J. Carey, D. J. DeWitt, M. J. Franklin, N. E. Hall, M. L. McAuliffe, J. F. Naughton, D. T. Schuh, M. H. Solomon, C. Tan, O. G. Tsatalos, et al. *Shoring up persistent applications*, volume 23. ACM, 1994.

[9] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *SoCC'10*, pages 143–154.

[10] J. C. Corbett and et al. Spanner: Google's Globally-Distributed Database. In *OSDI*, pages 251–264, 2012.

[11] C. Dennl, D. Ziener, and J. Teich. On-the-fly composition of fpga-based sql query accelerators using a partially reconfigurable module library. In *FCCM*, pages 45–52, 2012.

[12] C. Diaconu, C. Freedman, E. Ismert, P.-A. Larson, P. Mittal, R. Stonecipher, N. Verma, and M. Zwilling. Hekaton: SQL Server's memory-optimized OLTP engine. In *SIGMOD*, pages 1243–1254, 2013.

[13] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. The notions of consistency and predicate locks in a database system. *Commun. ACM*, 19(11):624–633, Nov. 1976.

[14] J. Evans. jemalloc. `http://canonware.com/jemalloc`.

[15] H. Garcia-Molina and K. Salem. Main memory database systems: An overview. *IEEE Trans. on Knowl. and Data Eng.*, 4(6):509–516, Dec. 1992.

[16] S. Ghemawat and P. Menage. TCMalloc: Thread-caching malloc. `http://goog-perftools.sourceforge.net/doc/tcmalloc.html`.

[17] J. Gray. *Concurrency Control and Recovery in Database Systems*, chapter Notes on data base operating systems, pages 393–481. Springer-Verlag, 1978.

[18] J. Gray. The transaction concept: Virtues and limitations. In *VLDB*, pages 144–154, 1981.

[19] J. Gray, P. Sundaresan, S. Englert, K. Baclawski, and P. J. Weinberger. Quickly generating billion-record synthetic databases. SIGMOD, pages 243–252, 1994.

[20] J. N. Gray, R. A. Lorie, G. R. Putzolu, and I. L. Traiger. Modelling in data base management systems. chapter Granularity of locks and degrees of consistency in a shared data base, pages 365–393. 1976.

[21] T. Haerder and A. Reuter. Principles of transaction-oriented database recovery. *ACM Comput. Surv.*, 15(4):287–317, Dec. 1983.

[22] S. Harizopoulos, D. J. Abadi, S. Madden, and M. Stonebraker. OLTP through the looking glass, and what we found there. In *SIGMOD*, pages 981–992, 2008.

[23] M. Heytens, S. Listgarten, M.-A. Neimat, and K. Wilkinson. Smallbase: A main-memory dbms for high-performance applications. Technical report, Hewlett-Packard Laboratories, 1995.

[24] R. Johnson and I. Pandis. The bionic dbms is coming, but what will it look like? In *CIDR*, 2013.

[25] R. Johnson, I. Pandis, N. Hardavellas, A. Ailamaki, and B. Falsafi. Shore-MT: a scalable storage manager for the multicore era. EDBT, pages 24–35, 2009.

[26] R. Johnson, I. Pandis, R. Stoica, M. Athanassoulis, and A. Ailamaki. Aether: a scalable approach to logging. *Proc. VLDB Endow.*, 3(1-2):681–692, 2010.

[27] H. Jung, H. Han, A. D. Fekete, G. Heiser, and H. Y. Yeom. A scalable lock manager for multicores. In *SIGMOD*, pages 73–84, 2013.

[28] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. Zdonik, E. P. C. Jones, S. Madden, M. Stonebraker, Y. Zhang, J. Hugg, and D. J. Abadi. H-Store: A High-Performance, Distributed Main Memory Transaction Processing System. *Proc. VLDB Endow.*, 1(2):1496–1499, 2008.

[29] H. T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *ACM Trans. Database Syst.*, 6(2):213–226, June 1981.

[30] P.-A. Larson, S. Blanas, C. Diaconu, C. Freedman, J. M. Patel, and M. Zwilling. High-performance concurrency control mechanisms for main-memory databases. *VLDB*, 5(4):298–309, Dec. 2011.

[31] J. Miller, H. Kasture, G. Kurian, C. Gruenwald, N. Beckmann, C. Celio, J. Eastep, and A. Agarwal. Graphite: A distributed parallel simulator for multicores. In *HPCA*, pages 1–12, 2010.

[32] D. L. Mills. Internet time synchronization: the network time protocol. *Communications, IEEE Transactions on*, 39(10):1482–1493, 1991.

[33] I. Pandis, R. Johnson, N. Hardavellas, and A. Ailamaki. Data-oriented transaction execution. *Proc. VLDB Endow.*, 3:928–939, September 2010.

[34] I. Pandis, P. Tözün, R. Johnson, and A. Ailamaki. PLP: Page Latch-free Shared-everything OLTP. *Proc. VLDB Endow.*, 4(10):610–621, July 2011.

[35] A. Pavlo, C. Curino, and S. Zdonik. Skew-aware automatic database partitioning in shared-nothing, parallel OLTP systems. In *SIGMOD*, pages 61–72, 2012.

[36] D. Porobic, I. Pandis, M. Branco, P. Tzn, and A. Ailamaki. OLTP on Hardware Islands. *Proc. VLDB Endow.*, 5:1447–1458, July 2012.

[37] K. Ren, A. Thomson, and D. J. Abadi. Lightweight locking for main memory database systems. In *VLDB*, pages 145–156, 2013.

[38] M. Rosenblum, E. Bugnion, S. A. Herrod, E. Witchel, and A. Gupta. The impact of architectural trends on operating system performance. In *SOSP*, pages 285–298, 1995.

[39] M. Stonebraker, S. Madden, D. J. Abadi, S. Harizopoulos, N. Hachem, and P. Helland. The end of an architectural era: (it's time for a complete rewrite). In *VLDB*, pages 1150–1160, 2007.

[40] S. S. Thakkar and M. Sweiger. Performance of an OLTP application on symmetry multiprocessor system. In *ISCA*, pages 228–238, 1990.

[41] The Transaction Processing Council. TPC-C Benchmark (Revision 5.9.0). `http://www.tpc.org/tpcc/spec/tpcc_current.pdf`, June 2007.

[42] P. Tözün, B. Gold, and A. Ailamaki. OLTP in wonderland: where do cache misses come from in major OLTP components? In *DaMoN*, 2013.

[43] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden. Speedy transactions in multicore in-memory databases. In *SOSP*, 2013.

[44] A. Whitney, D. Shasha, and S. Apter. High Volume Transaction Processing Without Concurrency Control, Two Phase Commit, SQL or C++. In *HPTS'97*.

[45] L. Wu, A. Lottarini, T. K. Paine, M. A. Kim, and K. A. Ross. Q100: the architecture and design of a database processing unit. In *ASPLOS*, 2014.