The Case for Determinism in Database Systems

Alexander Thomson
Yale University
thomson@cs.yale.edu

Daniel J. Abadi
Yale University
dna@cs.yale.edu

ABSTRACT

Replication is a widely used method for achieving high availability in database systems. Due to the nondeterminism inherent in traditional concurrency control schemes, however, special care must be taken to ensure that replicas don't diverge. Log shipping, eager commit protocols, and lazy synchronization protocols are well-understood methods for safely replicating databases, but each comes with its own cost in availability, performance, or consistency.

In this paper, we propose a distributed database system which combines a simple deadlock avoidance technique with concurrency control schemes that guarantee equivalence to a predetermined serial ordering of transactions. This effectively removes all nondeterminism from typical OLTP workloads, allowing active replication with no synchronization overhead whatsoever. Further, our system eliminates the requirement for two-phase commit for any kind of distributed transaction, even across multiple nodes within the same replica. By eschewing deadlock detection and two-phase commit, our system under many workloads outperforms traditional systems that allow nondeterministic transaction reordering.

1. INTRODUCTION

Concurrency control protocols in database systems have a long history of giving rise to nondeterministic behavior. They traditionally allow multiple transactions to execute in parallel, interleaving their database reads and writes, while guaranteeing equivalence between the final database state and the state which would have resulted had transactions been executed in some serial order. The key modifier here is *some*. The agnosticism of serialization guarantees to *which* serial order is emulated generally means that this order is never determined in advance; rather it is dependant on a vast array of factors entirely orthogonal to the order in which transactions may have entered the system, including:

- thread and process scheduling
- buffer and cache management
- hardware failures
- variable network latency
- deadlock resolution schemes

Nondeterministic behavior in database systems causes complications in the implementation of **database replication** and **horizontally scalable distributed databases**. We address both of these issues in turn.

1.1 Replication

Replication of OLTP databases serves several purposes. First, having multiple replicas of a database improves availability, since transactions can continue to be executed by other replicas should one go down. Furthermore, bringing a crashed server back up following a failure can be simplified by copying a replica's state instead of rebuilding the crashed server's state from logs [16]. Finally, read-only queries may be executed by only one replica, so replication can significantly improve throughput under read-heavy workloads.

The consequence of using nondeterministic concurrency control protocols is that two servers running exactly the same database software with the same initial state and receiving identical sequences of transaction requests may nonetheless yield completely divergent final database states. Replication schemes take precautions to prevent or limit such divergence. Three properties are desirable in a replication scheme:

- consistency across replicas
- currentness of all replicas
- low overhead

Since modern database systems allow nondeterministic behavior, replication schemes typically make tradeoffs between the properties of consistency, currentness, and low overhead. Commonly used replication schemes generally fall into one of three families, each with its own subtleties, variations, and costs.

- **Post-write replication.** Writes are performed by a single replica first, and the replication occurs after the write is completed. This category includes traditional master-slave replication, where all transactions are executed by a primary "master" system, whose write sets are then propagated to all other "slave" replica systems, which update data in the same order so as to guarantee convergence of their final states with that of the master. This is typically implemented via log shipping [14, 20]—the master sends out the transaction log to be replayed at each replica.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Articles from this volume were presented at The 36th International Conference on Very Large Data Bases, September 13-17, 2010, Singapore.
Proceedings of the VLDB Endowment, Vol. 3, No. 1
Copyright 2010 VLDB Endowment 2150-8097/10/09... $ 10.00.

This category also includes schemes where different data items have different masters, and variations on this theme where different nodes can obtain "leases" to become the master for a particular data item. In these cases, transactions that touch data spanning more than one master require a network communication protocol such as two-phase commit to ensure consistency across replicas. Distributed deadlock must also be detected if locking-based concurrency control protocols are used.

For both the traditional master-slave, and variations with different data being mastered at different nodes, writes occur at the master node first, and data is replicated after the write has completed. The degree to which replicas lag behind the master is dependent on the speed with which they apply the master's write sets, but they are always at least a small amount behind. Therefore, read-queries sent to replicas are not guaranteed to return fresh results without incurring the additional latency of waiting for replicas to "catch up" (although for some applications this is acceptable [22, 19, 2]).

Furthermore, there is a fundamental latency-durability-consistency tradeoff in post-write replication systems. Either latency is increased as the master node waits to commit transactions until receiving acknowledgment that shipped data has arrived at a replica ([15, 21]), or if not, then when the master fails, log records in flight at the time of the failure may not be delivered. In such a case, either the in-flight transactions are lost, reducing durability, or they are retrieved only after the failed node has recovered, but the transactions executed on the replica in the meantime threaten consistency.

- **Active replication with synchronized locking.** All replicas have to agree on write locks granted to data items [4]. Since writes can only proceed with an agreed-upon exclusive lock, all replicas will perform updates in a manner equivalent to the same serial order, guaranteeing consistency. The disadvantage of this scheme is the additional latency due to the network communication for the lock synchronization. For this reason, it is used much less frequently in practice than post-write replication schemes.

- **Replication with lazy synchronization.** Multiple active replicas execute transactions independently—possibly diverging temporarily—and reconcile their states at a later time [10, 7, 17]. Lazy synchronization schemes enjoy good performance at the cost of consistency.

If a database system were to execute sequences of incoming transactions in a purely deterministic manner, tradeoffs between the desirable properties described above could be avoided entirely. Transactions could be ordered in advance by a centralized server (or by a distributed service [18, 24]), dispatched in batches to each replica to be deterministically executed, assuring that each replica independently reaches a final state consistent with that of every other replica while incurring no further agreement or synchronization overhead.

1.2 Horizontal scalability

Horizontal scalability—the ability to partition DBMS data across multiple (typically commodity) machines ("nodes") and distribute transactional execution across multiple partitions while maintaining the same interface as a single-node system—is an increasingly important goal for today's database system designers. As a larger number of applications require a transactional throughput larger than what a single machine is able to deliver cost-effectively, the need for horizontally scalable database systems increases. However, non-deterministic behavior significantly increases the complexity and reduces the performance of ACID-compliant distributed database system designs.

In order to satisfy ACID's guarantees, distributed transactions generally use a distributed commit protocol, typically two-phase commit, in order to gather the commit/abort decisions of all participating nodes, and to make sure that a single decision is reached—even while participating nodes may fail over the course of the protocol. The overhead of performing these two phases of the commit protocol—combined with the additional time that locks need to be held—can significantly detract from achievable transaction throughput in partitioned systems. This performance hurdle has contributed to the movement towards using technologies that relax ACID (particularly atomicity or consistency) in order to achieve scalable distributed database systems (e.g. many so-called "NoSQL" systems).

Deterministic database systems that use active replication only need to use a one-phase commit protocol. This is because replicas are executing transactions in parallel, so the failure of one replica does not affect the final commit/abort decision of a transaction. Hence, the additional care that is taken in two-phase commit protocols to ensure appropriate execution in the face of potential node failure is not necessary. For transactions that have no user-level aborts specified in the transactional code (i.e. there exists no potential for integrity constraint violations or other conditional abort logic), the need for a distributed commit protocol is eliminated completely (since unpredictable events such as deadlock are barred from causing transactions to be nondeterministically aborted). Employing a deterministic execution scheme therefore mitigates, and, for some transactions, completely eliminates the most significant barrier to designing high-performance distributed transactional systems.

1.3 Contribution

In this paper, we present a transactional database execution model with the property that any transaction T_n's outcome is uniquely determined by the database's initial state and a universally ordered series of previous transactions $\{T_0, T_1, ..., T_{n-1}\}$ (Section 2).

Further, we implement a database system prototype using our deterministic execution scheme alongside one which implements traditional execution and concurrency control protocols (two-phase locking) for comparison. Benchmarking of our initial prototype, supported by analytical modeling, shows that the performance tradeoff on a single system (without considering replication) between deterministic and nondeterministic designs has changed dramatically with modern hardware. The deterministic scheme significantly lowers throughput relative to traditional schemes when there are long delays in transaction processing (e.g. due to fetching a page from disk). However, as transactions get shorter and less variable in length, the deterministic scheme results in nearly negligible overhead (Sections 3 and 4). We therefore conclude that the design decision to allow nondetermin-

ism in concurrency control schemes should be revisited.

We also examine the performance characteristics of our deterministic execution scheme when implemented in conjunction with the partitioning of data across multiple machines (Section 5). We find that our prototype outperforms systems relying on traditional lock-based concurrency control with two-phase commit on OLTP workloads heavy in multi-partition transactions.

2. MAINTAINING EQUIVALENCE TO A PREDETERMINED SERIAL ORDER

Since the nondeterministic aspects of current systems stem from the looseness of ACID's serializability constraints, to achieve entirely predictable behavior we restrict valid executions to those equivalent to a single predetermined serial execution.

The simplest way for a concurrency control protocol to guarantee equivalence to an execution order $\{T_0, T_1, ..., T_n\}$ would be to remove all concurrency, executing transactions one-by-one in the specified order. On modern multi-core machines, however, schemes which allow most processors to sit idle clearly represent suboptimal allocation of computational resources and yield poor performance accordingly.

Fortunately, it is possible to use locking to allow for some amount of concurrency while still guaranteeing equivalence to a predetermined serial order. Predetermined serial order equivalence (as well as deadlock freedom) can be achieved by restricting the set of valid executions to only those satisfying the following properties:

- **Ordered locking.** For any pair of transactions T_i and T_j which both request locks on some record r, if $i < j$ then T_i must request its lock on r before T_j does[1]. Further, the lock manager must grant locks strictly in the order that they were requested.
- **Execution to completion.** Every transaction that enters the system must go on to run to completion—either until it commits or until it aborts *due to deterministic program logic*. Therefore, if a transaction is delayed in completing for any reason (e.g. due to a hardware failure within a replica), that replica *must* keep that transaction active until the transaction executes to completion or the replica is killed[2]—even if other transactions are waiting on locks held by the blocked one.

In practice, ordered locking is typically implemented by requiring transactions to request all their locks immediately upon entering the system, although there exist transaction classes for which this may not be possible. We examine these cases in depth in Section 4.2. For the purposes of the comparisons presented in the next section, however, we consider a very straightforward implementation of the above scheme.

[1]This is a well-known deadlock avoidance technique. Postgres-R [13] is an example of a system that performs locking in this way.

[2]In some situations—for example when a transaction has deterministically entered a stalled state—it may be desirable to temporarily switch to a traditional execution and replication scheme. The prospect of a seamless, on-the-fly protocol for shifting between execution models presents an intriguing future avenue of research.

3. THE CASE FOR *NON*DETERMINISM

Before making the case for disallowing nondeterministic behavior in database systems, it is important to revisit the arguments for its inclusion.

A good transactional database system should be fast, flexible, and fault-tolerant. A premium is placed on the capability of transactional systems to guarantee high isolation levels while optimally allocating computational resources. It is desirable also to support essentially arbitrary user-defined transactions written in a rich and expressive query language.

Historically, solutions to many of the challenges of designing such systems have relied heavily on the loose serializability constraints discussed above. To illustrate the value of transaction reordering in achieving these goals, we consider a hypothetical transactional database system chugging along under some archetypical transactional workload. When a transaction enters the system, it is assigned to a thread which performs the transaction's task, acquiring locks appropriately, and then commits the transaction, releasing the locks it acquired. Let's say this hypothetical system is well-designed, with an impeccable buffer-management scheme, an efficient lock manager, and high cache locality, so that most transactions are executed extremely quickly, holding their locks for minimal duration. Lock contention is low and transactions complete almost as soon as they enter the system, yielding excellent throughput.

Now, suppose a transaction enters the system, acquires some locks, and becomes blocked for some reason (examples of this include deadlock, access to slow storage, or, if our hypothetical database spans multiple machines, a critical network packet being dropped). Whatever the cause of the delay, this is a situation where a little bit of flexibility will prove highly profitable. In the case of deadlock or hardware failure, it might be prudent to abort the transaction and reschedule it later. In other cases, there are many scenarios in which resource allocation can be vastly improved by shuffling the serial order to which our (non-serial) execution promises equivalence. For example, suppose our system receives (in order) a sequence of three transactions whose read and write sets decompose into the following:

T_0: read(A); write(B); read(X);

T_1: read(B); write(C); read(Y);

T_2: read(C); write(D); read(Z);

If T_0 becomes delayed when it tries to read X, T_1 will fail to acquire a read lock on record B and will be unable to proceed. In a deterministic system, T_2 would get stuck behind T_1 due to the read-write dependency on C. However, if we can modify the serial order to which we promise equivalence, T_2 could acquire its lock on C (since T_1 blocked on B before requesting it) and would complete its execution, moving ahead of T_0 and T_1 in the equivalent serial order. Therefore, as long as the system requires equivalence only to *some* execution order, and not necessarily to the order in which transactions were received, idle resources can immediately be effectively allocated to executing T_2.

The above example provides some insight into the class of problems that on-the-fly transaction reordering remedies. To better quantify this advantage, we create an analytical model and perform an experiment in which we observe the effects of introducing a stalled transaction in a tradi-

tional system vs. under the deterministic execution model described above. Due to space constraints, the analytical model is presented only in the appendix, but the model yields results consistent with those of the experiments presented here.

In our experiment, we implement a simple workload, where each transaction accesses 10 of a database's 10^6 records. After each relevant record is looked up in a secondary index, a lock is acquired and the item is updated. The transactions are fairly short: we measure that locks are released approximately $30\mu s$ after the last one is granted. Execution proceeds under favorable conditions until, at time=$1s$, a transaction enters the system, acquires 10 locks, and stalls completely for a full second. At time=$2s$, the stalled transaction releases its locks and commits. We measure throughput of each system as a function of time, as well as probability that new transactions entering the system will be unable to execute to completion due to lock contention. Under the deterministic locking scheme, all 10 of a transaction's locks are requested immediately when it enters the system, while in the traditional system, locks are requested sequentially, and execution halts upon a single lock acquisition failure. The traditional scheme also implements a timeout-based deadlock detector which periodically aborts and restarts any transaction (besides the initial slow transaction) which fails to make progress for a specified timeout interval. See the appendix for more on experimental setup.

Figure 1 shows our observations of the clogging behavior for several different contention rates, displaying both the probability that incoming transactions will be unable to immediately complete and the total transactional throughput as a function of time. Three key behaviors are evident here:

- **Comparable performance absent long transactions.** As long as all transactions complete in a timely manner, there's very little difference in performance between deterministic and nondeterministic transaction ordering schemes, regardless of read/write set conflict rates. In fact, it turns out that the traditional system actually executes and commits transactions in almost exactly the same order as the deterministic system during this period.

- **Relative sensitivities to stalled transactions.** When a transaction stalls one second into execution, the deterministic system becomes quickly clogged with transactions which can neither complete (due to lock contention) nor abort and restart later. In the traditional cases, where all locks are not immediately requested, and when other nondeterministic transaction reordering mechanisms are implemented, performance degrades far more gradually. The plateaus that we see in lock contention probability—which are reached much faster when contention is higher—result from a saturation of the system's (ample) threadpool with blocked transactions.

  In both systems, sensitivity to clogging and the severity of its effect depend on the conflict rate between transactions. Many modern OLTP workloads have low contention rates.

- **Clog resolution behavior.** Regardless of execution model, when the stalled transaction finally completes and releases its locks, the clog in the lock manager dissipates almost instantaneously, and throughput springs back to its pre-clog levels.

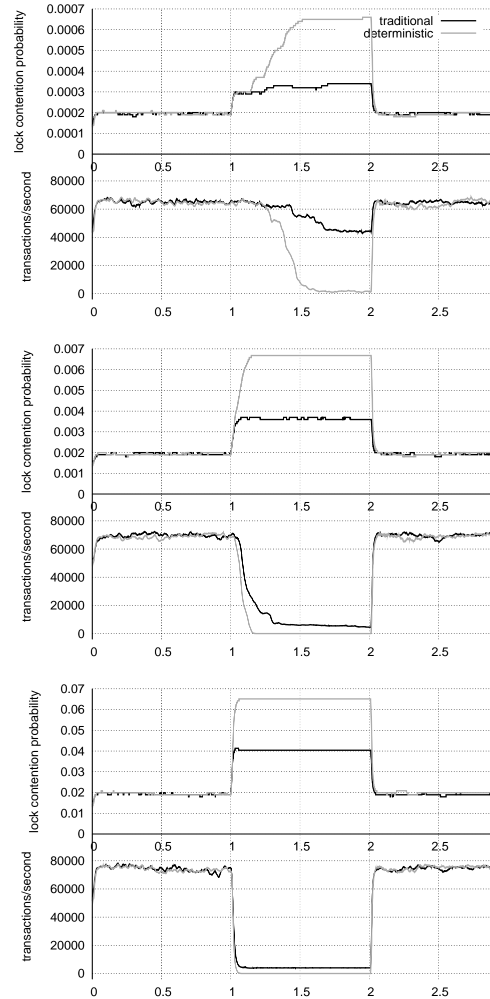

Figure 1: Measured probability of lock contention and transaction throughput with respect to time in a 3-second interval. Two transactions conflict with probabilities 0.01%, 0.1% and 1%, respectively.

From this experiment it is clear that where transactions can deadlock or stall for any reason, execution schemes which guarantee equivalence to an a priori determined serial order prove to be a poor choice. Hence, in an era where disk reads caused wild variation in transaction length, allowing nondeterministic reordering was the only viable option.

4. A DETERMINISTIC DBMS PROTOTYPE

As a larger percentage of transactional applications fit entirely in main memory; hardware gets faster, cheaper and more reliable; and OLTP workloads are dominated more and more by short, streamlined transactions implemented via stored procedures; we expect to find with diminishing frequency scenarios which would cause clogging in a deterministic system. The problem can be further ameliorated with a few simple insights:

- **Taking advantage of consistent, current replication.** Instantaneous failover mechanisms in actively replicated database systems can drastically reduce the impact of hardware failures within replicas. Highly replicated systems can also help hide performance dips that affect only a subset of replicas.
- **Distributed read-only queries.** Read-only queries need only be sent to a single replica. Alternatively, a longer read-only query can often be split up across replicas to reduce latency, balance load, and reduce the clogging effect that it might cause if run in its entirety on a single replica. Of course, long read-only queries are increasingly avoided by today's transactional application designers—instead they are often sent to data warehouses.

4.1 System architecture

Our deterministic database prototype consists of an incoming transaction preprocessor, coupled with arbitrarily many database replicas.

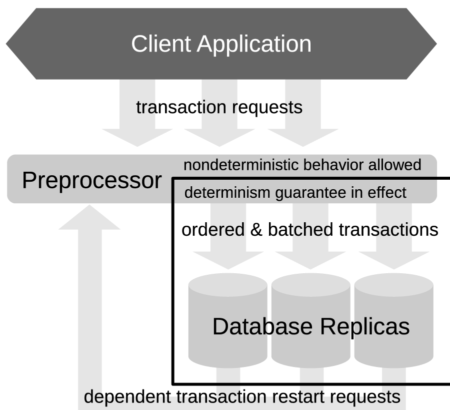

Figure 2: Architecture of a deterministic DBMS.

The preprocessor is the boundary of the system's internal deterministic universe. It accepts transaction requests, orders them, and performs in advance any necessary nondeterministic work (such as calls to `sys.random()` or `time.now()` in the transaction code), passing on its results as transaction arguments. The transaction requests are then batched and synchronously recorded to disk, guaranteeing durability. This is the pivotal moment at which the system is committed to completing the recorded transactions and after which all execution must be consistent with the chosen order. Finally, the batched transaction requests are broadcast to all replicas using a reliable, totally-ordered communication layer.

Each database replica may consist of a single machine or partition data across multiple machines. In either case, it must implement an execution model which guarantees both deadlock freedom and equivalence to the preprocessor's universal transaction ordering. The partitioned case will be discussed further in Section 5.

Upon the failure of a replica, recovery in our system is performed by copying database state from a non-faulty replica. Alternative schemes are possible (such as replaying the transactions from the durable list of transactions at the preprocessor), as long as the recovery scheme adheres to the system's determinism invariant.

4.2 Difficult transaction classes

It isn't necessarily possible for every transaction to request locks on every record it accesses immediately upon entering the system. Consider the transaction

$$
\begin{aligned}
&U(x):\\
&\quad y \leftarrow \mathrm{read}(x)\\
&\quad \mathrm{write}(y)
\end{aligned}
$$

where x is a record's primary key, y is a local variable, and write(y) updates the record whose primary key is y.

Immediately upon entering the system, it is clearly impossible for a transaction of type U to request all of its locks (without locking y's entire table), since the execution engine has no way of determining y's value without performing a read of record x. We term such transactions *dependent transactions*. Our scheme addresses the problem of dependent transactions by decomposing them into multiple transactions, all of which but the last work towards discovering the full read/write set so that the final one can begin execution knowing everything it has to access. For example, U can be decomposed into the transactions:

$$
\begin{aligned}
&U_1(x):\\
&\quad y \leftarrow \mathrm{read}(x)\\
&\quad \mathrm{newtxnrequest}(U_2(x,y))
\end{aligned}
$$

and

$$
\begin{aligned}
&U_2(x,y):\\
&\quad y' \leftarrow \mathrm{read}(x)\\
&\quad \mathrm{if}\ (y' \neq y)\\
&\quad\quad \mathrm{newtxnrequest}(U_2(x,y'))\\
&\quad\quad \mathrm{abort}()\\
&\quad \mathrm{else}\\
&\quad\quad \mathrm{write}(y)
\end{aligned}
$$

U_2 is not included in the ordered transaction batches that are dispatched from the preprocessor to the replicas until the result of U_1 is returned to the preprocessor (any number of transactions can be run in the meantime). U_2 has some information about what it probably has to lock and immediately locks these items. It then checks if it locked the correct items (i.e., none of the transactions that ran in the meantime changed the dependency). If this check passes, then U_2 can proceed; however, if it fails, then U_2 must be be aborted (and its locks released). The preprocessor is notified of the abort and includes U_2 again in the next batch of transactions that are dispatched to the replicas. Note that all abort-and-retry actions are deterministic (the transactions that ran between U_1 and U_2 will be the same across all replicas, and since the rescheduling of U_2 upon an abort is performed by the preprocessor, all future abort-and-retries are also deterministic).

Since U's decomposition requires only one additional transaction to calculate the full read/write set, we call U a first-order dependent transaction. First-order dependent transactions are often seen in OLTP workloads in the form of index lookups followed by record accesses. Higher-order dependent transactions such as the second-order transaction

$$
\begin{aligned}
&V(x):\\
&\quad y \leftarrow \mathrm{read}(x)\\
&\quad z \leftarrow \mathrm{read}(y)\\
&\quad \mathrm{write}(z)
\end{aligned}
$$

appear much less frequently in real-world workloads, but

the same decomposition technique handles arbitrary higher-order transactions.

This method works on a principle similar to that of optimistic concurrency control, and as in OCC, decomposed dependent transactions run the risk of starvation should their dependencies often be updated between executions of the decomposed parts.

To better understand the applicability and costs of this decomposition technique, we perform a series of experiments and support them with an analytical model. Full details of the experiments and model are included in the appendix. We observed that performance under workloads rich in first-order dependent transactions is inversely correlated with the rate at which the decomposed transactions' dependencies are updated. For example, in a workload consisting of TPC-C Payment transactions (where a customer name is often supplied in lieu a primary key, necessitating a secondary index lookup), throughput will suffer noticeably only if every single customer name is updated extremely often—hundreds to thousands of times per second. The overhead of adding the additional read transaction to learn the dependency is almost negligible. Since real-life OLTP workloads seldom involve dependencies on frequently updated data (secondary indexes, for example, are not usually created on top of volatile data), we conclude that workloads that have many dependencies do not generally constitute a reason to avoid deterministic concurrency control.

This scheme also fits nicely into database system environments that allow users to adjust the isolation level of their transaction in order to improve performance. This is because there is a straightforward optimization that can be performed for dependent reads that are being run at the read-committed isolation level (instead of the fully serializable isolation level). The transaction is still decomposed into two transactions as before, but the second no longer has to check to see if the previously read data is still accurate. This check (and potential abort) are therefore eliminated. Note that database systems implementing traditional two-phase locking also struggle with high contention rates inherent to workloads rich in long read-only queries, and that many such systems already support execution at reduced isolation levels[3]. We envision the potential for some back-and-forth between the deterministic database system and the application designer, where the application designer is alerted upon the submission of a transaction with a dependent read that performance might be improved if this transaction was executed at a lower isolation level.

5. TPC-C & PARTITIONED DATA

To examine the performance characteristics of our deterministic execution protocol in a distributed, partitioned system, we implement a subset of the TPC-C benchmark consisting of 100% New Order transactions (the backbone of the TPC-C benchmark). The New Order transaction simulates a customer placing an e-commerce order, inserting several records and updating stock levels for 5-15 items (out of 100000 possible items in each warehouse).

Figure 3 shows throughput for deterministic and traditional execution of the TPC-C New Order workload, varying frequency of multipartition transactions (in which part of the customer's order must be filled by a warehouse on a remote node). In these experiments, data is partitioned across two partitions. We include measurements for 2 warehouses and 10 warehouses (1 per partition and 5 per partition, respectively). See the appendix for further discussion of experimental setup.

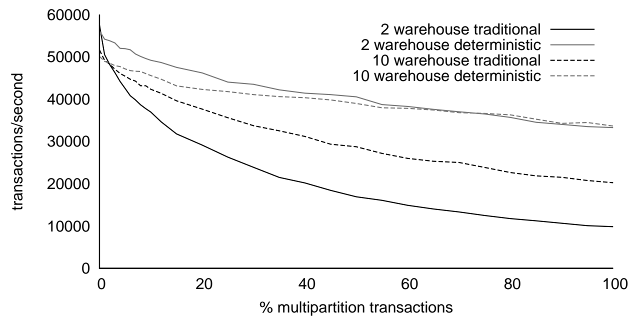

Figure 3: Deterministic vs. traditional throughput of TPC-C (100% New Order) workload, varying frequency of multipartition transactions.

When multipartition transactions are infrequent, New Order transactions stay extremely short, and the two execution schemes yield comparable throughput for a given data set size—just as we observed in the experiment in Section 3 when no anomalously slow transactions were present in the execution environment. With only 2 warehouses, both schemes enjoy better cache locality than with 10 warehouses, yielding a improved throughput absent multipartition transactions. The fewer records there are, however, the more lock conflicts we see. Two New Order transactions conflict with probability approximately 0.05 with 2 warehouses and approximately 0.01 with 10 warehouses. Under both systems, the overall decline in performance as a larger percentage of transactions become multipartition is therefore greater with 2 warehouses than with 10 (since multipartition transactions increase transaction length, exacerbating the effect of lock conflict).

When we compare transactional throughput under the two execution models, one might expect the clogging behavior discussed in Section 3 to sink the deterministic scheme's performance compared to that of traditional execution when network delays begin to entail longer-held locks—especially in the 2-warehouse case where lock contention is very high. In fact, we see the opposite: regardless of number of warehouses (and therefore contention rate) the deterministic protocol's performance declines more gracefully than that of traditional locking as multipartition transactions are added.

This effect can be attributed to the additional time that locks are held during the two-phase commit protocol in the traditional execution model. In the traditional case, all locks are held for the full duration of two-phase commit. Under deterministic execution, however, our preprocessor dispatches each new transaction to every node involved in processing it. The transaction fragment sent to each node is annotated with information about what data (e.g. remote reads and whether conditional aborts occur) is needed from which other nodes before the updates involved in this fragment can "commit". Once all necessary data is received from all expected nodes, the node can safely release locks for

[3]Multiversion and snapshot systems do not find read-only queries problematic, but these systems are orthogonal the approach described here, since there is room for multiversion implementations of deterministic database systems.

the current transaction and move on the the next transaction. No commit protocol is necessary to ensure that node failure has not affected the final commit/abort decision since node failure is a nondeterministic event, and nondeterministic events are not allowed to case a transaction to abort (the transaction will have committed on a replica, which can be used for recovery of the failed node as mentioned above).

This means that for multipartition transactions in the deterministic concurrency control case, each node working on a transaction needs to send at most one message to each other node involved in the transaction—and, conversely, needs to receive at most one message from any other node. As long as all nodes involved in a multipartition transaction emit their communications at roughly the same time (which frequently happens in our experiments since multi-partition transactions naturally cause synchronization events in the workload), locks are held for an additional duration of only a single one-way network message.

Under the traditional execution model, each incoming transaction which blocks on a multipartition New Order must remain blocked for up to the full duration of two-phase commit—itself compounding the clog and further increasing likelihood of lock contention for subsequent incoming transactions. Measurements of actual lock contention and average transaction latency taken during these experiments are presented in the appendix and indicate that greater duration of distributed transactions leads both to greater lock contention than we see under the deterministic scheme and to a higher cost associated with each instance of lock contention.

6. RELATED WORK

Database systems that provide stricter serializability guarantees than equivalence to some serial order have existed for over three decades. For example, there have been proposals for guaranteeing equivalence to a given timestamp order using a type of timestamp ordering concurrency control known as conservative T/O [3] (later referred to as "ultimate conservative T/O" [5] to distinguish it from other forms of timestamp ordering concurrency control that allow transaction aborts and reordering, thereby providing weaker serializability guarantees). Conservative T/O delays the execution of any database operation until it can be certain that all potentially conflicting operations with lower timestamps have already been executed. In general, this is done by having a scheduler wait until it receives messages from all possible transaction sources, and then scheduling the lowest timestamped read or write operation from all sources.

This naive implementation is overly conservative, since in general it serializes all write operations to the database. However, additional concurrency can be obtained by using the transaction class technique of SDD-1 [6] where transactions are placed into transaction classes, and only potentially conflicting transaction classes need to be dealt with conservatively. This is facilitated when transactions declare their read and write sets in advance. Our work builds on these decades-old techniques for guaranteeing equivalence to a single serial order; however we chose to implement our deterministic database using locking techniques and optimistic methods for dealing with the lack of knowledge of read/write sets in advance since most modern database systems use locking-based techniques for concurrency control instead of timestamp-based techniques.

The theoretical underpinnings for the observation that deterministic execution within each replica facilitates active replication can be found in work by Schneider [23]. This work shows that if each replica receives transactions in the same order and processes them deterministically in a serial fashion, then each replica will remain consistent. Unfortunately the requirement that each replica executes using a single thread is not a realistic scenario for highly concurrent database systems. This observation was also made by Jimenez-Peris et. al. [11], so they introduced a deterministic thread scheduler enabling replicas to execute transactions using multiple threads; each thread is scheduled identically on each replica (with careful consideration for dealing with the interleaving of local thread work with work resulting from messages received from other servers as part of distributed transactions). Our work differs from this work by Jimenez-Peris et. al. since we allow threads to be scheduled arbitrarily (giving the scheduler more flexibility and allowing network messages to be processed immediately instead of waiting until local thread work is complete before processing them) and guarantee determinism through altering the database system's concurrency control protocol.

Recent work by Basile et. al. [1] extends the work by Jimenez-Peris et. al. by intercepting mutex requests invoked by threads before accessing shared data, increasing concurrency by allowing threads that do not access the same data to be scheduled arbitrarily. This solution, however, requires sending network messages from leader nodes to replicas to order mutex acquisition across conflicting threads, whereas our solution does not require concurrency control network messages between replicas.

The idea to order transactions and grant write locks according to this order in order to facilitate eager replication was presented in work by Kemme and Alonso [13]. Our work differs from theirs in that our technique does not use shadow copies and only requires a reliable, totally-ordered group communication messaging protocol in sending requests from the preprocessor to the database—never within multipartition transactions, as this would cause locks to be held for prohibitively long durations in a deterministic system. Furthermore we handle dependent transactions using an optimistic method.

The observation that main memory database systems result in faster transactions and lower probability of lock contention was made by Garcia-Molina and Salem [9]. They further argue that in many cases it is best to execute transactions completely serially and avoid the overhead of locking altogether. Our work must deal with a more general set of assumptions relative to Garcia-Molina and Salem in that even though transactions in main memory are faster than transactions which might have to go to disk, we consider the case of network messages (needed for distributed transactions) increasing the length of some transactions. Furthermore, we do not require that transactions be executed serially (even though equivalence to a given serial order is guaranteed); rather multiple threads can work on different transactions in parallel.

Whitney et. al. [26], Pacitti et. al [18], Stonebraker et. al.[24], and Jones et. al. [12] all propose performing transactional processing in a distributed database without concurrency control by executing transactions serially in a single thread on each node (where a node in some cases can be a single CPU core in a multicore server [24]). Whit-

ney et. al. does not do this to perform active replication (updates to replicas are logged first), but the latter two papers take advantage of the determinism to perform active replication. However, multi-node (distributed) transactions are problematic in both cases. Our scheme takes advantage of deterministic concurrency control to avoid two-phase commit, which significantly reduces the cost of multi-node transactions, and further provides high levels of concurrency across multiple threads.

Middleware solutions such as that implemented by xkoto (recently acquired by Teradata) and Tashkent-API [8] attempt to perform replication over multiple database systems by applying the same transactions in the same order on each system. However, since the database systems do not guarantee equivalence to any given serial order, the middleware systems reduce the level of concurrency of transactions sent to the database systems to avoid replica divergence, potentially reducing throughput. Our solution is native to the database system, greatly reducing necessary middleware complexity.

Tay et. al. compare traditional dynamic locking and static locking (where all locks in a transaction are acquired immediately) using a detailed analytical model [25]. However, for static locking, if all locks are not able to be acquired immediately, the transaction is aborted and restarted (and hence the protocol is not deterministic). Furthermore, the model does not deal with the case where the location of all data that needs to be locked is not known in advance.

7. CONCLUSIONS

We revisited in this paper the decision to allow nondeterministic behavior in database systems.

We have shown that in light of current technology trends, a transactional database execution model which guarantees equivalence to a predetermined serial execution in order to produce deterministic behavior is viable for current main memory OLTP workloads, greatly facilitating active replication. Deterministic systems also render two-phase commit unnecessary in distributed database systems, thus removing performance barriers to distributed database systems.

8. ACKNOWLEDGMENTS

We are extremely appreciative of Phil Bernstein, Azza Abouzeid, Evan P. C. Jones, and Russell Sears for reading early versions of this paper and graciously contributing keen observations and helpful suggestions and corrections.

This material is based in part upon work supported by the National Science Foundation under Grant Number IIS-0845643.

9. REFERENCES

[1] C. Basile, Z. Kalbarczyk, and R. K. Iyer. Active replication of multithreaded applications. *IEEE Transactions on Parallel and Distributed Systems*, 17:448–465, 2006.

[2] P. A. Bernstein, A. Fekete, H. Guo, R. Ramakrishnan, and P. Tamma. Relaxed-currency serializability for middle-tier caching and replication. In *Proc. of SIGMOD*, pages 599–610, 2006.

[3] P. A. Bernstein and N. Goodman. Timestamp-based algorithms for concurrency control in distributed database systems. In *Proc. of VLDB*, pages 285–300, 1980.

[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, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[6] P. A. Bernstein, D. W. Shipman, and J. B. Rothnie, Jr. Concurrency control in a system for distributed databases (sdd-1). *ACM Trans. Database Syst.*, 5(1):18–51, 1980.

[7] Y. Breitbart, R. Komondoor, R. Rastogi, S. Seshadri, and A. Silberschatz. Update propagation protocols for replicated databates. *SIGMOD Rec.*, 28(2):97–108, 1999.

[8] S. Elnikety, S. Dropsho, and F. Pedone. Tashkent: uniting durability with transaction ordering for high-performance scalable database replication. In *Proc. of EuroSys*, pages 117–130, 2006.

[9] H. Garcia-Molina and K. Salem. Main memory database systems: An overview. *IEEE Transactions on Knowledge and Data Engineering*, 4(6), 1992.

[10] J. Gray, P. Helland, P. O'Neil, and D. Shasha. The dangers of replication and a solution. In *Proc. of SIGMOD*, pages 173–182, 1996.

[11] R. Jimenez-Peris, M. Patino-Martinez, and S. Arevalo. Deterministic scheduling for transactional multithreaded replicas. In *Proc. of IEEE Int. Symp. on Reliable Distributed Systems*, 2000.

[12] E. P. C. Jones, D. J. Abadi, and S. R. Madden. Concurrency control for partitioned databases. In *Proc. of SIGMOD*, 2010.

[13] B. Kemme and G. Alonso. Don't be lazy, be consistent: Postgres-r, a new way to implement database replication. In *Proc. of VLDB*, pages 134–143, 2000.

[14] R. P. King, N. Halim, H. Garcia-Molina, and C. A. Polyzois. Management of a remote backup copy for disaster recovery. *ACM Trans. Database Syst.*, 16(2):338–368, 1991.

[15] K. Krikellas, S. Elnikety, Z. Vagena, and O. Hodson. Strongly consistent replication for a bargain. In *ICDE*, pages 52–63, 2010.

[16] E. Lau and S. Madden. An integrated approach to recovery and high availability in an updatable, distributed data warehouse. In *Proc. of VLDB*, pages 703–714, 2006.

[17] E. Pacitti, P. Minet, and E. Simon. Fast algorithms for maintaining replica consistency in lazy master replicated databases. In *Proc. VLDB*, pages 126–137, 1999.

[18] E. Pacitti, M. T. Ozsu, and C. Coulon. Preventive multi-master replication in a cluster of autonomous databases. In *Euro-Par*, pages 318–327, 2003.

[19] C. Plattner and G. Alonso. Ganymed: scalable replication for transactional web applications. In *Proc. of Middleware*, pages 155–174, 2004.

[20] C. A. Polyzois and H. García-Molina. Evaluation of remote backup algorithms for transaction-processing systems. *ACM Trans. Database Syst.*, 19(3):423–449, 1994.

[21] C. A. Polyzois and H. García-Molina. Evaluation of remote backup algorithms for transaction-processing systems. *ACM Trans. Database Syst.*, 19(3):423–449, 1994.

[22] U. Röhm, K. Böhm, H.-J. Schek, and H. Schuldt. Fas: a freshness-sensitive coordination middleware for a cluster of olap components. In *Proc. of VLDB*, pages 754–765, 2002.

[23] F. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv.*, 22(4), 1990.

[24] M. Stonebraker, S. R. 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 *Proc. of VLDB*, 2007.

[25] Y. C. Tay, R. Suri, and N. Goodman. A mean value performance model for locking in databases: the no-waiting case. *J. ACM*, 32(3):618–651, 1985.

[26] A. Whitney, D. Shasha, and S. Apter. High volume transaction processing without concurrency control, two phase commit, SQL or C++. In *HPTS*, 1997.

APPENDIX

Prototype implementation & experimental setup

Our prototype is implemented in c++. All transactions are hand-coded inside the system. All benchmarks are taken on a Linux cluster comprised of quad-core Intel Xeon 5140 machines, each with 4GB of 667 MHz FB-DIMM DDR2, connected over a 100 megabit, full duplex local area network. At no point during data collection for any of the experiments performed in this paper did we observe network bandwidth becoming a bottleneck.

Clogging experiment details

Index lookups entail traversing B+ trees, while records are directly addressed by their primary keys. Updated records are chosen with a Gaussian distribution such that any pair of transactions will conflict on at least one data item with easily measurable and variable probability.

As the reader may notice in Figure 1, regardless of execution scheme, peak throughput is actually greater in the higher-conflict-rate trials than in lower-contention cases. Increased skew in the record-choice distribution results in higher lock contention rates by making certain records more likely to be accessed by all transactions, but this also has the effect of increasing cache locality, and therefore has a positive effect on throughput when no clog is present.

TPC-C New Order experiment details

In this experiment, we distribute data over two partitions on separate physical machines within our cluster, with a measured network message round-trip latency averaging $130\mu s$. We dedicate one core to query execution at each partition. All measurements are taken over a 10-second interval following a short ramp-up period.

Performance model of the effects of clogging

We consider a database system executing a workload homogeneously composed of a transaction consisting of n updates, where any two updates are disjoint with probability s (i.e. they conflict with probability $1-s$). If we run a transaction T_i when k locks are held by other transactions, we expect T_i immediately to acquire all n locks with probability

$$p_n = (s^k)^n = s^{kn}$$

In this case, we assume (since transactions are short) that T_i commits and releases its locks immediately, so that T_{i+1} also executes in an environment where k records are locked.

If T_i accesses any of the k locked records, however, it will be unable to complete. In this case, we wish to examine the expected number of new records T_i locks to determine the change in k. In a traditional execution model, T_i requests locks as it goes, blocking upon the first conflict and requesting no further locks. Expected change in k in the traditional case therefore depends on the number of locks that are acquired (represented by m below) *before* one overlaps with one of the k already-held locks:

$$\Delta k \;=\; \sum_{m=0}^{n-1} m p_m(\bar{p}_1)$$

where

$$p_x = s^{kx} \quad \text{and} \quad \bar{p}_x = (1-s^k)^x$$

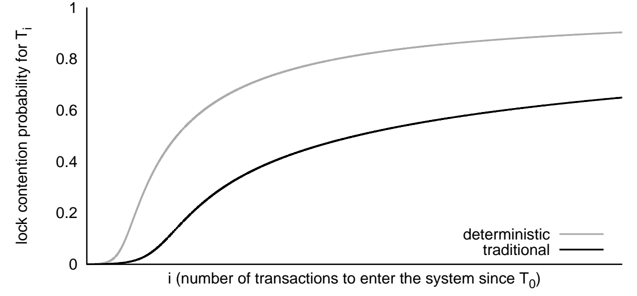

Figure 4: Probability that each new transaction entering the system conflicts with currently held locks.

Once again p_x represents the probability that a set of x lock requests all succeed; $\bar{p}_x$ represents the probability that a set of x lock requests all fail.

With our deterministic execution model, we have T_i request all locks upon beginning execution, regardless of which are granted, so expected change in k depends on the total number of records on which T_i conflicts with the k prior held locks ($n-m$ below, since m is again the number of new locks acquired):

$$\Delta k \;=\; \sum_{m=0}^{n-1} m p_m(\bar{p}_{n-m})\binom{n}{m}$$

In both cases, we are interested not simply in how many records are locked, but what effect this has on subsequent execution. Figure 4 depicts the quantity $1-p_n$ (the probability that incoming transactions will be *unable* to execute due to a lock conflict) as a function of i (the number of new transactions which have entered the system since T_0).

The origin represents the point in execution immediately following T_0 stalling: $i=0$ and $1-p_n = 1-s^{n^2}$. With both execution models, early on—before many transactions have gotten blocked directly or indirectly on T_0—most transactions are able to execute without experiencing lock contention, so k grows slowly. Further on, more transactions get stuck, clogging the lock manager and creating a positive feedback loop in which the clog grows exponentially for some time before asymptotically approaching a 100% chance that incoming transactions will be unable to complete. Unsurprisingly, the clog explodes much earlier in the deterministic case where even transactions that conflict on very few of their updates add their full read/write sets to the set of locked records.

Since the purpose of this model is not to predict performance characteristics of actual systems but simply to explain clogging behavior in general, we omit any horizontal scale and observe that for any s, n, and initial value of k, the model will yield a graph of this shape, albeit compressed and/or shifted. Two further shortcomings of this simple model make it unsuitable to fit to experimental data: (a) it does not take into account active reordering (via releasing locks and restarting slow transactions besides the stalled one) in the traditional execution case, and (b) the horizontal axis measures how many new transactions have entered the system since T_0 stalled, which is non-linearly correlated with time, especially when transactions get periodically restarted and when execution is performed by a fixed-size thread pool.

Performance model of decomposed first-order dependent transactions

In order to identify and classify the situations in which the method of transaction decomposition described in Section 4.2 is effective, we introduce to this discussion the notion of a record's *volatility*, which we define as the frequency with which a given record is updated.

Under workloads heavy in decomposed dependent transactions, we expect good performance if the transactions' dependencies are not often updated. As the records on which transactions in the workload depend become more volatile, however, performance is expected to suffer.

Let (T_1, T_2) represent the decomposition of a first-order dependent transaction T whose read/write set depended on a set of tuples $\mathbb{S}$. The total time during which an update to a record $r \in \mathbb{S}$ could cause T_2 to have to abort and restart is approximately equal the time between T_1 initiating the transaction request for T_2, and T_2 getting started. We will refer to this time as D. If R represents total transaction throughput and V represents the volatility of $\mathbb{S}$, then the probability that no transaction updates tuple during any given interval of length D is given by

$$P = (1 - V/R)^{DR}$$

and the expected number of times T_2 will be executed is

$$\sum_{i=0}^{\infty} P(1-P)^i(i-1) \; = \; 1/P$$

Figure 5 describes the expected number of times a typical decomposed transaction needs to be restarted as a function of the volatility of its dependencies.

Experimental measurement of decomposed first-order dependent transactions

To confirm the above result experimentally, we implement the decomposition of a simple first-order dependent transaction isomorphic to U in Section 4, which performs a lookup in a secondary index and then updates a record identified by the result. As a baseline, we also implement a transaction which performs the same task (an index lookup followed by a record update), but in a non-dependent fashion (i.e. with its full read/write set supplied as an argument). We then execute a variable mix of these two transactions while a separate, dedicated processor performs a variable number of updates per second on each entry in the index, redirecting that entry to identify a different record.

Figure 6 shows total transactional throughput and the number of times a transaction must be restarted on average before executing to completion—both as a function of the average volatility of index entries. Results are included for workloads consisting of 0%, 20%, 40%, 60%, 80% and 100% dependent transactions.

As one would expect, the workload consisting purely of non-dependent transactions is essentially unaffected by frequency of index updates, while more dependent workloads loads fare worse as volatility increases.

We observe, however, that even in highly dependent workloads, when volatility is reasonably low to moderate (under 1000 updates per second of each record), decomposing transactions to achieve compute read/write sets has almost negligible impact on performance.

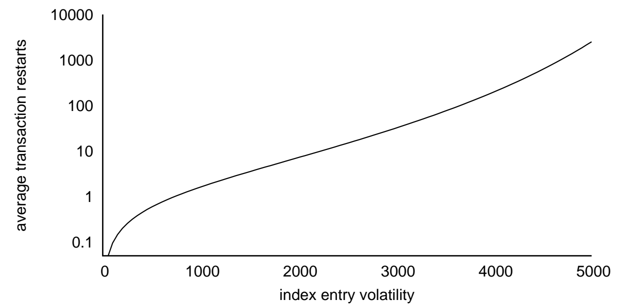

Figure 5: Expected number of times a decomposed first-order dependent transaction must be restarted as a function of total volatility of its dependencies.

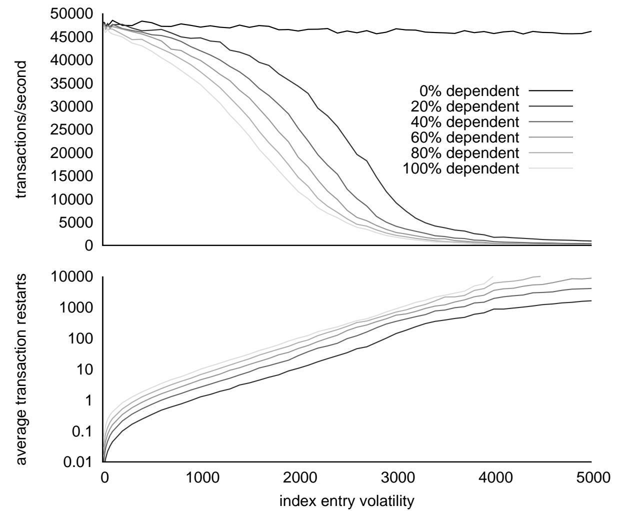

Figure 6: Measured throughput and average number of restarts of first-order dependent transactions as a function of volatility of their dependencies.

Additional TPC-C performance characteristics

To support the analysis of the performance advantage of avoiding two-phase commit in partitioned databases (presented in Section 5), we include in Figure 7 two additional measurements generated from the same experiment that was run to generate Figure 3: chance of lock contention, and average transaction latency (we include the original Figure 3 showing transaction throughput at the top of Figure 7 for visual convenience). Lock contention is measured as the total fraction of transactions which blocked at least once due to failure to acquire a lock. Transaction latency is measured from when a transaction begins executing at a replica until the time it commits at that replica.

Plotting contention and latency reveals two important effects which help illustrate the details of what is happening in our experiment:

- **The cost of two-phase commit.** In the presence of multipartition transactions, traditional execution suffers much worse latency than deterministic execution. The resulting longer-held locks give rise to a corresponding increase in measured lock contention.

- **Relative costs of increased contention.** Decreasing the total number of warehouses from ten to two results in increased contention and latency under both execution schemes. The impact of this on throughput is very clearly visible in the traditional case. In the deterministic case, this is not immediately obvious, since the two-warehouse version of the benchmark seems to outperform the ten-warehouse version for most of the graph. However, the improved cache locality of the two-warehouse case is hiding what is truly going on. A more careful examination of the graph will observe a steeper decline in throughput in the two warehouse case relative to ten warehouse case as the percentage of multipartition transactions increases. Hence the increased contention of the two warehouse case does indeed affect throughput for the deterministic implementation. Overall, we see that increasing the contention rate has *qualitatively* similar effects on the performance profiles of the two schemes—but due to higher transaction latency, the traditional execution is hit much harder by the added contention.

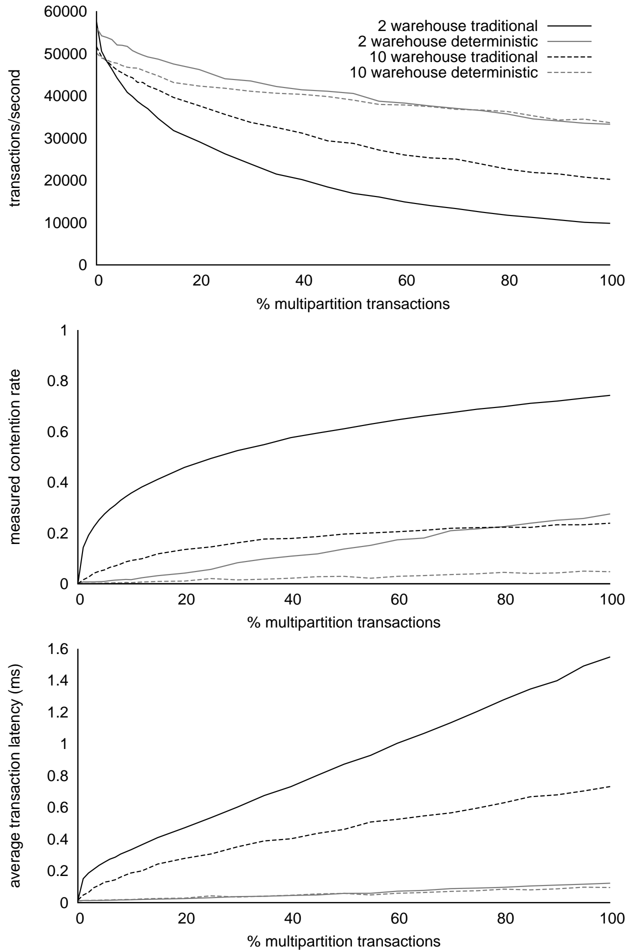

Figure 7: TPC-C (100% New Order) transactional throughput, measured lock contention, and average transaction latency.

Future Work

We intend to pursue several future avenues of research:

Preprocessor reordering

While arbitrary transaction reordering during execution is likely to break the determinism invariant, the initial choice of transaction order to which to guarantee equivalence is much freer. It is not hard to see that if a local transaction request arrives immediately after a distributed transaction conflict, contention will ensue if the order is preserved, but can probably be reduced if the local transaction is moved before the distributed one. We plan to investigate reordering techniques to be implemented in the preprocessor, so as to further reduce workload contention levels and expand the set of workloads for which a deterministic execution model would be viable.

Seamless transitions between deterministic and nondeterministic execution/replication models

As short as update transactions tend to be in today's OLTP workloads, and as reliable as we expect a deterministic system to be given adequate replication, there may be times where it is still prudent to nondeterministically reorder or abort transactions in some replicas. This obviously cannot occur unchecked, as such a scenario would lead to inconsistencies between replicas. We therefore intend to examine the possibility of supporting other traditional replication methods—inter-replica commit protocols, eventual consistency, and/or log-shipping methods—in such a way that the system can seamlessly and dynamically switch back and forth between traditional and deterministic execution and replication models, depending on the present workload and other variable conditions.

Thread scheduling in deterministic systems

In traditional systems, transactions block on only one lock acquisition at a time. Worker thread scheduling therefore tends to be pretty easy: when you free a contended resource, simply wake up the thread which is acquiring it. When transactions request all locks simultaneously and may be blocked on multiple contentions, this may not always be the best solution. Our experimentation thus far has demonstrated that minimizing superfluous context switches (i.e. avoiding prematurely scheduling threads blocked on multiple contentions) is key to maintaining acceptable performance in deterministic systems in the face of higher rates of lock contention. Our current prototype uses timer-based and randomized techniques to accomplish reasonable thread scheduling, but we believe further investigation in this area could yield better scheduling with less overhead.

Other deterministic concurrency control schemes

The lock-based deterministic system proposed in this paper guarantees equivalence to a predetermined serial ordering of transactions, but there are certainly other mechanisms by which such a property can be guaranteed. It would be interesting to develop, compare, and benchmark deterministic variants of other concurrency control schemes—particularly multiversion concurrency control.