An Evaluation of the Advantages and Disadvantages of Deterministic Database Systems

Kun Ren
Northwestern Polytechnical University, China
renkun_nwpu@mail.nwpu.edu.cn

Alexander Thomson
Google
agt@google.com

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

ABSTRACT

Recent proposals for deterministic database system designs argue that deterministic database systems facilitate replication since the same input can be independently sent to two different replicas without concern for replica divergence. In addition, they argue that determinism yields performance benefits due to (1) the introduction of deadlock avoidance techniques, (2) the reduction (or elimination) of distributed commit protocols, and (3) light-weight locking. However, these performance benefits are not universally applicable, and there exist several disadvantages of determinism, including (1) the additional overhead of processing transactions for which it is not known in advance what data will be accessed, (2) an inability to abort transactions arbitrarily (e.g., in the case of database or partition overload), and (3) the increased latency required by a preprocessing layer that ensures that the same input is sent to every replica. This paper presents a thorough experimental study that carefully investigates both the advantages and disadvantages of determinism, in order to give a database user a more complete understanding of which database to use for a given database workload and cluster configuration.

1. INTRODUCTION

There have been several recent proposals for database system architectures that use a deterministic execution framework to process transactions [9, 7, 24, 25, 26, 27]. Deterministic execution requires that the database processes transactions in a way that guarantees that if the database system is given the same transactional input, it will always end in the same final state. This is a much stronger guarantee than traditional database ACID guarantees, which guarantee only that the database system will process transactions in a manner that is equivalent to some serial order (but different instances of the database system can process the same set of input transactions in a way that is equivalent to two different serial orders).

Proceedings of the VLDB Endowment, Vol. 7, No. 10

Historically, the main advantages that these proposals for deterministic database systems attempt to achieve are related to replication and high availability — in deterministic systems, no communication whatsoever is required between replicas to keep them consistent (they are guaranteed not to diverge if they receive the same input).

The main disadvantage of deterministic transaction execution that is most commonly cited is the reduced processing flexibility that results from the stronger guarantees that deterministic systems need to make. When a thread that is processing a transaction stalls (e.g., due to a need to wait until a page on disk is brought into the buffer pool, or a need to wait for the next command from a user), a deterministic database system has less choice about what other transactions it can run in order to do useful work during the stall. This effectively reduces concurrency, which can lead to lower transactional throughput and longer latencies.

However, as more and more database systems are becoming "in-memory" (most of the working data set can be kept in memory), and user stalls are becoming increasingly rare in modern applications, the reduced executional flexibility of deterministic database systems is becoming less of a burden. Consequently, the above cited proposals for deterministic database systems argue that the advantages of deterministic database systems now outweigh the disadvantages.

Unfortunately, the tradeoff is not so simple, and deterministic database systems have both additional advantages and disadvantages not mentioned above. In addition to the replication advantage, deterministic databases have several additional advantages:

- A side-effect of the more constrained processing choices is deadlock avoidance — deterministic databases never have to worry about deadlock detection or aborting transactions due to deadlock.
- Nondeterministic events such as node failure cannot cause a transaction to abort (since different replicas will not observe the same set of nondeterministic events). Rather, active replica nodes need to step in on the fly for failed nodes (or, alternatively, the input log is replayed from a checkpoint to create a new node that had the same deterministic state of the failed node at the time of failure). Therefore, commit protocols for distributed transactions (such as two phase commit) that check for node failure before transaction commit, can be significantly simplified (or even entirely eliminated in same cases).

On the other hand, in addition to the lack of execution flexibility disadvantage, deterministic databases have the following disadvantages:

- Deterministic database systems either do not allow concurrency (to avoid nondeterministic events resulting from thread scheduling) or only allow concurrency if the system knows exactly what data will be accessed before the transaction begins concurrent execution. The former option significant reduces execution flexibility, while the latter option requires either overhead from the user to rewrite applications to annotate transactions with data access information, or an automatic process to discover the read-write set in advance (which incurs certain overhead).
- Transactions cannot be arbitrarily aborted without some sort of agreement protocol across replicas. At times of high database load when transactions may need to be aborted in order to prevent the negative side-effects of system overload (and performing agreement protocols across replicas is particularly difficult in times of high system load), deterministic database system performance suffers.
- In order for multiple replicas to receive the same input, there needs to be a layer above the database system that receives transactions from all clients and forwards these transactions (usually in batches) to the database system. If there is enough transactional input such that this preprocessing layer must include more than one machine, then an agreement protocol must be performed between the machines in this layer in order to deterministically merge the input. Although it does not reduce throughput, this preprocessing layer does increase latency.

Given these advantages and disadvantages, it is unclear when a deterministic database system should be used, and when a more traditional architecture should be used. Although the papers that introduced proposals for deterministic database system architectures tend to list (a subset of) these advantages and disadvantages, the experimental studies of these papers tend to focus on the advantages, with little investigation of the disadvantages.

Therefore, the primary contribution of this paper is a more complete experimental study of the advantages and disadvantages of determinism, so that database users and designers can get a better sense of which database architectures to use for a given target workload. We vary many workload parameters such as data contention, size of the database cluster, percentage of distributed transactions, heterogeneity of the cluster, percentage of transactions where the data that will be accessed is known in advance, and transaction complexity/length or order to gain a thorough understanding of the significance and impact of the above mentioned advantages and disadvantages.

As expected, the experimental study concludes that different database architectures are appropriate for different situations. However, some of our results are surprising. We found that the inability of deterministic database systems to abort transactions in times of overload is a much larger disadvantage than the requirement to derive in advance all items that will be locked. On the other hand, the deadlock avoidance advantage of deterministic database systems is by far their greatest advantage for achieving scalable transactional performance.

2. BACKGROUND

In this section we give some background on deterministic database systems and describe the important differences from traditional database systems.

2.1 Transaction Processing

As mentioned in the introduction, deterministic database systems must guarantee that they end in only one possible final state after processing a set of input transactions. Assuming the set of input transactions are arranged in a sequence and nondeterministic code inside transactions (such as calls to RAND or TIME) has been replaced with hard-coded constants (this is usually done by a preprocessing layer), there have been two primary approaches to accomplishing deterministic execution:

The first approach is to execute transactions one at a time in this input sequence, since this eliminates nondeterminism stemming from concurrency control [21]. (Nondeterministic failures are dealt with by replaying the input transaction log from a checkpoint to recover state at the time of failure). Variations on this scheme allow concurrency by dividing both the data and transactional workload into disjoint partitions, and allowing each partition to run their local partitions in serial order[1] (which can be done straightforwardly if there are no multi-partition transactions) [24, 25, 10].

The second approach allows for increased concurrency by allowing transactions (even inside the same partition) to be processed in parallel, but carefully restricting lock acquisition such that locks are acquired in order of a transaction's location in the input sequence. This approach ensures that the resulting equivalent serial order is equal to the input sequence, and also ensures that there is no deadlock. For distributed implementations (where data is partitioned across different nodes), each database node is provided the same view of the log of the global transaction input, and ensures that it acquires its locks in the correct order for any local or distributed transactions in the log that it is involved in.

Unfortunately, if a local transaction comes before a distributed transaction in the global order, a node must acquire locks for the local transaction before the distributed transaction. Therefore, other nodes that require a remote read from a slow node must wait until it completes all conflicting local transactions ahead of the current transaction before it can acquire the read lock and send the data to the faster node. This effectively results in the entire cluster processing transactions at the same speed as the slowest node if there are even a moderate number of distributed transactions involving the slow node. If there are faster replicas of the slow node, read-only queries can be sent to the faster replica, which helps to alleviate this problem; however, if there are no read-only queries or no faster replicas, the system will always run at the speed of the slowest node.

Nondeterministic systems will also observe throughput slowdown if there is a continuous stream of distributed transactions involving a slow node; however, they have more recourse to deal with such an issue (e.g., aborting or reordering local transactions).

2.2 Concurrency

Another disadvantage of acquiring locks in transaction order is reduced transactional scheduling flexibility. The next transaction in the transaction order cannot begin execution until all of the locks of the previous transaction have been requested. Therefore, while nondeterministic systems are

[1]In some cases transactions can be executed out of order using speculative execution techniques [8]

allowed to request locks for transactions in parallel, deterministic systems must serialize this process, potentially resulting in a new bottleneck. Furthermore, in order to reduce the overhead of the lock request process and to allow transactions later in the input sequence to get started, deterministic systems typically request all locks for a transaction in a single batch quickly at the beginning of the transaction.

In practice, this means that the transaction needs some way to know in advance all items that it will need to access, so that it can make all the needed requests at the beginning of the transaction. For many transactions, especially those in which records are accessed through a primary key, a static analysis of the transaction request is sufficient to deduce which records will be accessed. However, records accessed through a secondary index are problematic for static analysis and other techniques must be used.

Thomson et. al. propose an optimistic protocol, called OLLP, for determining which records need to be locked [26]. The basic idea is to do a test run for transactions that can not be statically analyzed. The test run does not write any data to the database — it just performs enough of the transaction to get a sense of what records are accessed. The transaction is then annotated with the records that were accessed, and locks are requested for these records when the transaction begins to be officially processed. In some cases, the set of records that the transaction actually needs to access are different than the records accessed in the trial run (e.g., if there was an update to the secondary index in the meantime). In that case, each replica will run into the same problem (since they all have the same deterministic view of the database state at the time a transaction begins) and they will each independently decide to abort the transaction and restart it with a new set of lock requests.

This optimistic protocol to handle transactions that cannot be statically analyzed automates the process of deducing in advance which records will be accessed by a transaction. However, it adds latency (the time to do the trial run) and reduces throughput (the trial run is done by the same worker nodes that are processing official transactions, so it consumes limited resources). Therefore, workloads with many transactions that fit into the category of being unable to be statically analyzed are potentially problematic for deterministic database systems.

2.3 Agreement on Input

Another disadvantage of deterministic database systems is the need to have global agreement on the sequence of input transactions. Whether the deterministic system processes transactions serially or whether it uses the lock acquisition protocol described above, the order of transactions that the system must guarantee serializable equivalence to must be agreed upon across all nodes within and across replicas. There have been several proposals in the literature for how to do this:

- Have one machine that accepts all incoming transactions [26, 30]. This machine collects all incoming transactions and broadcasts them (in batches) to each database node. The machine actively replicates its state to a hot-backup to avoid being a single-point of failure.
- Allow any node in the cluster to accept transactions, and when the node receives a transaction, it is immediately given a timestamp based on the local clock of the node [24]. The concatenation of the local timestamp with the node id provides a global timestamp. Transactions are forwarded to relevant nodes, which wait a certain delay and then execute all transactions in order of global timestamp. Tuning this delay appropriately is critical — if the delay is too short, it is possible to receive transactions with a timestamp that precedes a transaction that the node has already executed, and if the delay is too long, transactions have high latency.
- Have a preprocessing layer that receives incoming transactions and runs an agreement protocol that concatenates transactions into a input transaction log which is forwarded to the database cluster [28].

Of these approaches, the first approach can only scale to the rate one machine can receive network messages with transactional input, the second approach may result in cascading aborts and other significant problems stemming from delayed network messages, and the third approach results in additional transactional latency due to the agreement protocol in the preprocessing layer. In practice, the first and third approaches are most commonly used[2], where the first approach is used for smaller scale deployments and the third approach is used for larger scale deployments.

2.4 Commit Protocols

Despite the long list of significant disadvantages of determinism described above, determinism does have several advantages. Perhaps the least obvious of these is the ability of deterministic database systems to shorten (or eliminate) distributed commit protocols. To understand why this is the case, consider that the two primary purposes of commit protocols are to (1) guarantee atomicity by ensuring that all nodes involved in processing a transaction are prepared to commit and (2) guarantee durability by ensuring that the results of a transaction have reached stable storage and that a failure of a node during the protocol will not prevent its ability to commit the transaction upon recovery.

Due to the differences in the way failures are handled in deterministic database systems, much of the effort of traditional commit protocols is unnecessary. Unlike a traditional database system where all transactions running on a failed node are aborted, deterministic database systems do not have this as an option, since failures are nondeterministic events, and replicas processing the same transactions at the same time may not fail. Therefore transactions running on a failed node are not aborted — they simply can not continue to be processed at that node until the node recovers.

The failed node recovers its state at the time of the crash by loading a checkpointed snapshot of database state, and replaying the input transaction log deterministically from that point [26, 28, 15, 14]. If replicas of this failed node remain active, then the rest of the database nodes do not need to wait for the failed node to recover — they can proceed with transaction processing and if they need data stored on the failed node as part of a distributed transaction, they can reroute that request to live replicas of the failed node that are processing transactions in parallel.

The key thing to note from this recovery process is that nondeterministic failure (no matter the reason for the failure, e.g., a failed node, corrupt memory, out-of-memory/disk,

[2]VoltDB recently switched from the second approach to a variant of the first where local transactions can sometimes avoid being sent to the central aggregator node [30].

Feature of Determinism	Advantage	Disadvantage
No nondeterministic aborts	Simplified commit protocols	Cannot arbitrarily abort transactions in times of overload or local problems such as out-of-memory/disk
Input transactions placed in sequence	Transaction sequence becomes redo log, simplifying recovery	Increased latency due to preprocessing layer that does the transaction sequencing
Acquires locks in transaction order	No deadlocks	Reduced concurrency

Table 1: Many distinguishing characteristics of determinism come with both advantages and disadvantages

etc.) will not result in a transaction being aborted, since the database can always recover by replaying the input transaction log in order to eventually commit a transaction (in the case of out-of-memory/disk, it may need to replay this log on a new/larger database server node). Therefore, a distributed commit protocol does not need to worry about ensuring that no node fails during the commit protocol, and it does not need to collect votes from nodes involved in the transaction if the only reason why they would vote against a transaction committing is due to node (or any other type of nondeterministic) failure. Put a different way: the only thing a commit protocol needs to check is whether there was any node that executed code that deterministically could cause an abort (e.g an integrity constraint being violated).

For transactions that do not contain code that could cause a transaction to deterministically abort, no commit protocol whatsoever is required in deterministic database systems. For transactions that *do* contain code that could result in a deterministic abort, nodes involved in those transactions can vote 'yes' as soon as they can be sure that they will not deterministically abort the transaction. Therefore, transactions do not need to wait until the end of processing before initiating the commit protocol.

2.5 Summary

In this section we have described the advantages and disadvantages of determinism. As summarized in Table 1, individual design decisions of deterministic database systems often lead simultaneously to benefits and performance hazards. The next section attempts to quantify these advantages and disadvantages, in order to give database designers and users a better sense of when deterministic database systems should be used, and when they should not be used.

3. EXPERIMENTAL EVALUATION

All the experiments measuring throughput were conducted on Amazon EC2 using m3.2xlarge (Double Extra Large) instances, which have 30GB of memory and 26 EC2 Compute Units–8 virtual cores with 3.25 Compute Units each. Experiments were run on a shared-nothing cluster of 8 of these Double Extra Large EC2 instances, unless stated otherwise. Although the EC2 virtual machines were usually similar in performance to each other, we did notice some variation. We discuss this phenomenon further in Section 3.8. We have made the source code we used for our experiments available at: https://github.com/yaledb/calvin.

3.1 Benchmarked Systems

Although there have been several proposals and implementations of deterministic databases over the past decade, we preferred to experiment with more recent code since getting decade-old code to compile and run on modern hardware can be challenging. The code for the H-Store and Calvin deterministic prototypes were both available to us; in the end we decided to use the Calvin codebase for our implementation, since the Calvin codebase has an option to turn off locking and process transactions using H-Store's completely serial execution (per-partition) model.

Furthermore, since Calvin has a fully functioning lock manager, we were able to reuse the lock manager code for the two-phase locking implementation in the traditional database prototype. This is important: we wanted to avoid an apples-to-oranges comparison as much as possible, so we went to great effort to build the traditional database implementation inside the Calvin codebase (reusing the same code for shared components such as client communications, thread handling, admission control, network messaging and handling, storage layer, etc). Therefore, the only difference between the two prototypes are the relevant details around deterministic vs. nondeterministic execution: the deterministic prototype has a preprocessing layer, a worker thread in charge of acquiring locks in the correct deterministic order, and code for running the optimistic lock prediction protocol, while the nondeterministic prototype has a two-phase locking implementation, deadlock detection and elimination, and two phase commit code. The prototype is implemented in C++.

Of the 8 cores on each EC2 instance, we devote 3 cores to the shared database components that are equivalent for both the deterministic and nondeterministic prototypes (e.g., client communications, inter-node communications, etc), and the remaining 5 cores are allocated to worker threads that process transactions in a deterministic or nondeterministic way.

3.1.1 Deterministic implementation

For the deterministic prototype we allocate one core to a lock acquisition thread and the remaining 4 cores to threads that actively process transactions. This is because the deterministic database system requires that locks are acquired in the correct order, and our implementation achieves this by only allowing one thread to perform lock acquisition for all transactions. Since no worker thread can proceed without acquiring its locks, we wanted to ensure that the lock acquisition thread has no competition for CPU resources, and therefore dedicated a core to this thread. Unfortunately, this means that when transactions are "long" and lock acquisition is a small percentage of actual transaction work, dedicating an entire core to lock acquisition is wasteful, and this core runs at far less than 100% utilization.

In order not to overlook the consequences of this design decision, we experiment with both "short" transactions that only perform one read/write action per each item that is locked (thereby resulting in the lock acquisition thread being

fully utilized, and in some cases, even being a bottleneck) and "long" transactions which perform a set of computations totaling 15 μs of CPU work for each record that is accessed. In practice this resulted in over 30% of transaction execution time being spent acquiring locks for "short" transactions (an unusually high number) and 16% of transaction execution time being spent acquiring locks for "long transactions" (a number that Harizopoulos et. al. report is typical in modern database systems on OLTP workloads [5]).

The Calvin prototype comes with two different lock managers: one that acquires and releases locks using a traditional hash-table based lock manager that tracks which transactions are waiting for which locks, and one that acquires locks using the VLL protocol [20] — a lighter-weight lock manager implementation for deterministic systems. We found that VLL only improved throughput over the traditional hash-table based lock manager when the lock acquisition thread described above is a bottleneck; otherwise the performance of both lock managers are nearly identical. Where relevant, we present results below for both lock managers; however, when the results are identical (or close to identical) we present results for just VLL.

3.1.2 Nondeterministic implementation

There have been many recent promising proposals for (nondeterministic) scalable transactional database systems [1, 3, 10, 12, 13, 18, 22, 29, 31]. These proposals are for complete system designs, and therefore differ from each other and from traditional database designs in many dimensions (not just determinism vs. nondeterminism). Furthermore some of these designs do not use 2PL-based approaches for concurrency control; for example, HANA uses MVCC [13], Hekaton uses optimistic MVCC [3], and Google F1 uses OCC (in addition to some pessimistic locking) [22]. Since deterministic versions of MVCC and OCC have not yet been proposed in the literature, it is impossible to do a direct comparison of deterministic vs. nondeterministic versions of these approaches. Therefore, we focus our comparisons on deterministic vs. nondeterministic lock-based concurrency control within a single prototype, as discussed above.

In contrast to the deterministic prototype (where one of the 5 worker cores is dedicated entirely to lock management), for the nondeterministic prototype, each of the 5 worker cores contain threads that are actively processing transactions. In our initial version of our nondeterministic prototype, we used a traditional thread-per-DBMS worker process model (according to the language of Hellerstein et. al. [6]), both with and without a thread pool. However, we found that with many distributed transactions, many threads were sitting idle waiting for network messages. When we used a thread pool, it was not uncommon for every thread in the pool to be idle waiting for messages, rendering the entire system idle until at least one of these messages arrive. In order to maximize throughput of our system, we had to use a very large thread pool, but this resulted in many threads being active, and the constant switching between them yielded a noticeable overhead that limited system throughput.

Therefore we allowed threads in the thread pool to work on multiple transactions at once; in this way there could be more active transactions than threads in the thread pool. We found this approach often yielded much higher maximum throughput than the traditional thread-per-DBMS process model. Each worker thread maintains a C++ struct, called ActiveTxnMap, that stores the context of all of the transactions assigned to it that are waiting for network messages. As soon as a transaction needs to block to wait for a network message, that transaction's context is placed in the ActiveTxnMap, and the thread starts working on a different transaction. When the network message arrives, the thread retrieves the transaction's context from the ActiveTxnMap and continues to work on that transaction. The lock manager also contains two C++ structs containing transaction context. The first, called BlockedTxnMap, contains transactions that are blocked, waiting to acquire locks. The lock manager continues to update the context of transactions in the BlockedTxnMap as they acquire locks over time; as soon as all locks for a transaction have been acquired, the transaction is moved from the BlockedTxnMap to the second struct maintained by the lock manager: the ReadyTxnQueue. Both the BlockedTxnMap and the ReadyTxnQueue are thread safe, and any worker thread can retrieve the context of a transaction from the ReadyTxnQueue and execute it (however, working on transactions in their own ActiveTxnMap that are now able to run take priority).

For the experiments in this paper, we allowed both the deterministic and nondeterministic prototypes to use either the traditional thread-per-worker process model or our more advanced process model, and selected the best results for each particular data point (in every case, both the deterministic and nondeterministic prototypes agree on the optimal process model for that data point, so differences in the process model do not affect our experimental results).

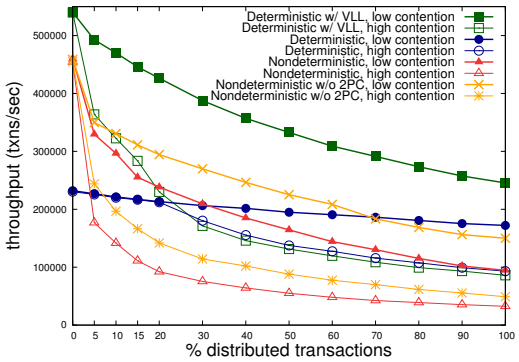
Although the deterministic prototype is guaranteed to be deadlock-free, the nondeterministic prototype can result in deadlock. We spent a long time experimenting with multiple different deadlock detection and elimination protocols. In general, while we found that it was possible to keep the overhead of deadlock detection and elimination low for deadlocks local to a single machine using timeout-based deadlock detection (optionally Dreadlocks optimizations can be used for local deadlock [11]), dealing with distributed deadlock is much more challenging due to the unpredictable wait time for remote messages. Timeout-based techniques do not work well for distributed deadlock, and therefore the wait-for graph implementation from Gray [4] and Stonebraker [23] remain the state of the art. We therefore used this implementation for distributed deadlock detection in the nondeterministic prototype.

The nondeterministic prototype uses traditional two phase commit for distributed transactions. However, in order to understand how much of a contribution the overhead of two phase commit adds to the results, and to account for proposals that optimize two phase commit in various ways, we also present results for what the nondeterministic prototype would be able to achieve if there were no commit protocol whatsoever[3]. Optimized two-phase commit implementations therefore fall somewhere between these two extremes.

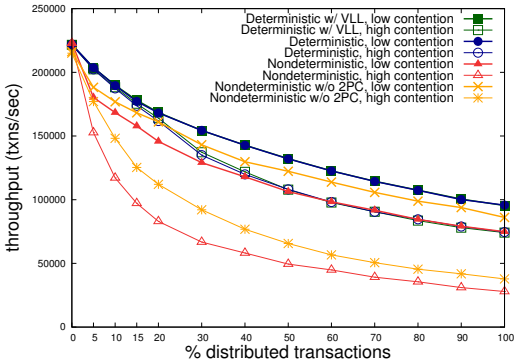
3.2 Benchmarks

Our goal in this paper is not to validate determinism as an execution strategy—rather, we want to classify the trans-

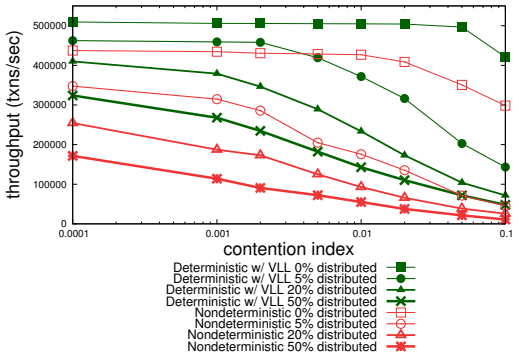
[3]We present these results to illustrate performance repercussions only—the resulting system does *not* preserve ACID semantics, as nondeterministic execution protocols rely on a distributed protocols to ensure atomicity for distributed transactions.



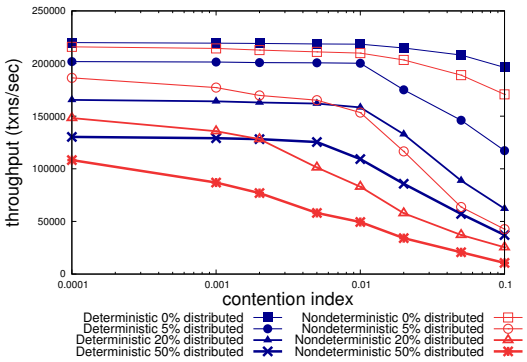
(a) Throughput of "short" transactions vs. frequency of distributed transactions.



(b) Throughput of "long" transactions vs. frequency of distributed transactions.



(c) Throughput of "short" transactions vs. contention index.



(d) Throughput of "long" transactions vs. contention index.

Figure 1: Microbenchmark throughput experiments.

actional workloads for which deterministic execution outperforms traditional methods and those for which it does not. Therefore, we created our own microbenchmarks that carefully test specific workload characteristics in a way that we can vary parameters for maximum flexibility. However, we also include results for the TPC-C benchmark since we modified the Calvin prototype to perform the experiments in this paper, but want to verify that our prototype still yields the same trends as previously published work (our results should be compared to an equivalent experiment published in VLDB 2013 [20]).

3.3 Microbenchmark Experiments

We use a simple microbenchmark to explore the effects of various parameters that are likely to have an effect on the determinism/nondeterminism tradeoff: likelihood of transactions to lock the same records, percentage of transactions that are distributed, number of partitions spanned that participate in each distributed transaction, and length of time transactions take to execute their internal logic.

Each EC2 instance in our shared-nothing cluster of machines contains a database partition consisting of 1,000,000 records. Each transaction reads and updates a number of records (10 records unless otherwise specified). Our microbenchmark does not contain read-only transactions because there is no difference between how our deterministic and nondeterministic prototypes handle read-only transactions — therefore we focus only on types of transactions where there is a difference between the systems.

In order to carefully vary the contention of the workload, we divide the data set into "hot records" and "cold records". Unless otherwise specified, each transaction accesses one hot record at each partition that participates in its execution; and all remaining accesses are to cold records. Cold record accesses have negligible contention, so the contention of any particular experiment can be finely tuned by varying the size of the hot record set. If there are K hot records at each partition, then the probability of any two transactions conflicting at that partition is $1/K$, since this is the probability that they will attempt to access the same hot record. We refer to this probability as the contention index (CI): 100 hot records per partition implies a contention index $CI = 0.01$, 10,000 implies a contention index of $CI = 0.0001$.

3.3.1 *Contention Experiments*

In Figures 1(a) and 1(b), we plot the maximum throughput that each system was able to obtain vs. the percentage of transactions that are distributed. Each of these plots shows measurements for $CI = 0.01$ ("high contention") and $CI = 0.0001$ ("low contention") for each of our four Calvin configurations: deterministic execution (with the default lock manager), deterministic execution with VLL, nondeterministic execution (with 2PC) and nondeterministic execution without 2PC. (Figures 1(c) and 1(d) vary contention index in order to get more than 2 data points for contention index).

As explained in the previous section, short transactions spend over 30% of transaction processing time acquiring

locks. This is sufficient for the 4 cores running worker threads to overwhelm the one core performing lock acquisition for the deterministic prototype, and lock acquisition therefore becomes a bottleneck. However, by reducing the overhead of the lock manager by a factor of 2.3, VLL eliminates this bottleneck. Therefore, there are significant differences in performance of the deterministic system for the two different lock managers when transactions are short. However, when transactions are longer, the lock manager thread is not a bottleneck in either case, and both deterministic systems perform similarly.

Nondeterministic execution, however, suffers from two performance setbacks compared to deterministic execution: (1) the overhead of (and increased contention caused by holding locks during) the two-phase commit protocol (2PC) and (2) the possibility of deadlocks. To understand the significance of the first factor (2PC), Figures 1(a) and 1(b) include a nondeterministic scheme that skips the two-phase commit protocol, illustrating the isolated throughput costs that nondeterministic execution schemes incur. From these figures, it is clear that the requirement to perform 2PC can significantly reduce throughput — sometimes by as much as as 30%. However, this factor alone does not explain the performance difference between the deterministic and nondeterministic prototype. Furthermore the reduction in throughput caused by 2PC counter-intuitively gets smaller with a higher percentage of distributed transactions and high contention. Clearly, distributed deadlocks are the more significant performance hazard for the nondeterministic system.

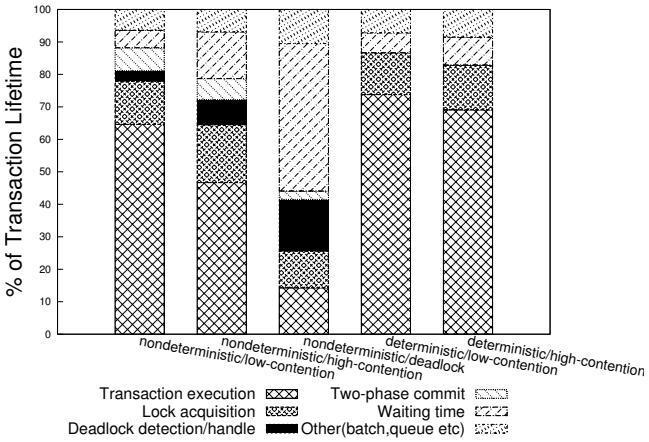


Figure 2: Performance breakdown of each variant.

To better understand the performance difference between the systems (and especially how distributed deadlocks effect throughput), we investigated how much time is spent in our prototype doing useful transaction work vs. how much time is spent (1) acquiring locks (2) detecting/handling deadlock (3) performing two-phase commit and (4) spinning with nothing to do (if all active transactions are either waiting for locks or waiting for remote data). Figure 2 presents the results of the performance breakdown for the data points from Figure 1(b) along the vertical axis of 20% distributed transactions. For the nondeterministic system we show three different breakdowns of how time is spent in the different components of the system. The first two show the performance of the system under low and high contention when distributed deadlock is not present in the system. The third bar in the figure shows how the performance breakdown changes (relative to the second bar) when distributed deadlock is present (distributed deadlock was very rare for the low contention workload, so we only show the breakdown for high contention).

As can be seen in Figure 2, the baseline deadlock detection/handling code is very small. When there is distributed deadlock in the system, this code becomes slightly most costly, but the main reason why the percentage of time spent doing useful transaction work plummets in the third bar in the figure is because of the increased "waiting time". This is caused by other transactions getting stuck behind deadlocked transactions. Any transaction whose lock set overlaps with a deadlocked transaction becomes blocked; any transaction whose lock set overlaps with either the deadlocked transaction or the newly blocked transaction just mentioned becomes blocked; and so on for subsequent transactions. Eventually most of the hot data in the database becomes locked and the database becomes clogged. As soon as the deadlock is discovered and the transaction is aborted, the database de-clogs. However, the database becomes clogged faster than the inherent delay involved in sending messages over the network to create the "waits-for" graph and discover the deadlock. Therefore, the average time the system spins with nothing to do increases sharply when distributed deadlock is present.

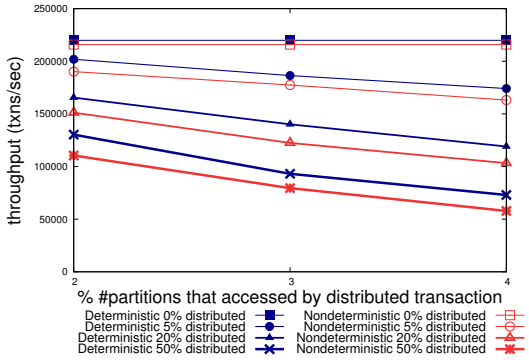
3.3.2 *Multi-node distributed transactions*

Past papers on deterministic systems experimented with distributed transactions. However, a transaction needs only to involve two nodes for it to be called a "distributed transaction", and indeed the distributed transaction experiments in previously published papers only involve two nodes. We therefore now run experiments on workloads in which distributed transactions span more than two nodes.

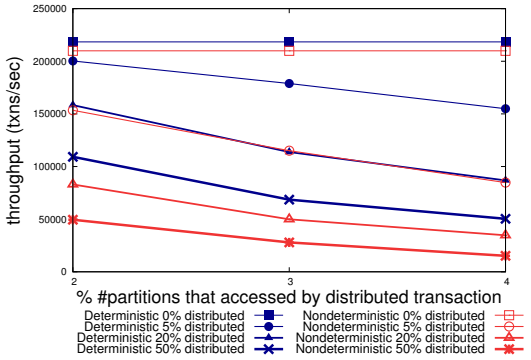
Remember that in our microbenchmark, each distributed transaction conflicts with other transactions with probability CI at *each* participating node. Therefore, a 4-node transaction conflicts with approximately twice as many other transactions as a 2-node transaction. We therefore expect to see a corresponding drop in throughput due to the higher contention.

Indeed, Figures 3(a) and 3(b) show how throughput is affected as we increase the number of nodes involved in distributed transactions, from 2 to 4 nodes. When there are no distributed transactions, throughput is unaffected, since it does not matter how many nodes are involved in a distributed transaction if they do not exist. The more distributed transactions, the greater the relative drop in throughput as those distributed transactions span more partitions. In the most extreme cases that we measured (50% distributed transactions), the step from two- to four-partition distributed transactions decreased throughput by a factor of approximately two for both systems.

The most important conclusion to draw from Figures 3(a) and 3(b) is that although throughput drops with more nodes involved in distributed transactions, the slope of the lines for the corresponding deterministic and nondeterministic pairs are the same. This indicates that these two factors — determinism and distributed transaction size — are independent of each other.



(a) Throughput vs. participants per distributed transaction. Long transactions, low contention (ci=0.0001).



(b) Throughput vs. participants per distributed transaction. Long transactions, high contention (ci=0.01).

Figure 3: Throughput vs. number of partitions accessed by distributed transactions (low and high contention).

3.4 TPC-C

For our TPC-C benchmark experiments, each EC2 instance contains 20 warehouses. In order to vary the percentage of distributed transactions, we vary the percentage of New Order and Payment transactions that access a remote warehouse. The average contention index (CI) for TPC-C is approximately 0.01.

Figure 4 shows the results of our experiments on the TPC-C benchmark. Since the transaction logic of TPC-C is complex, and the contention index is high, the results are similar to our microbenchmark experiments using long transactions under high contention. However, throughput drops less significantly (relative to our previous experiments) as the percentage of distributed transactions increase. This is because in our microbenchmarks, each distributed transaction touches one "hot" key per partition, whereas in TPC-C, each distributed transaction in TPC-C touches one "hot" key total (not per partition). Therefore with more distributed transactions, the contention index actually decreases.

When there are no distributed transactions, the nondeterministic system outperforms the deterministic system. This is because TPC-C transactions perform even more computation per item locked than the "long" transactions in the microbenchmark. This results in fewer locks requested per second, and thus reduced work for the lock acquisition thread. Our decision to devote an entire core to lock acquisition in the deterministic system is therfore costly in this case — this core is significantly underutilized, and potential CPU resources are wasted. In contrast, the nondeterministic system is able to fully utilize all CPU resources. Since the 2PC and distributed deadlock disadvantages of nondeterministic database systems are not present when there are no distributed transactions, the better CPU utilization of the nondeterministic system results in it outperforming the deterministic system. However, as the percentage of distributed transactions increase, the advantages of determinism become increasingly present.

3.5 Resource Constraints

Next, we explore the effects of variable resource availability on nondeterministic vs. deterministic systems. Here, we are trying to model the effects of server overloading, poor performance isolation between virtual machines, network flakiness, and heterogeneous hardware environments—

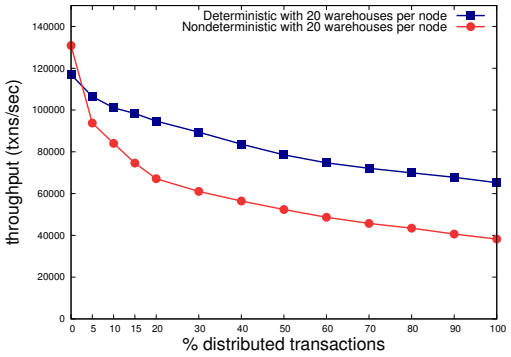


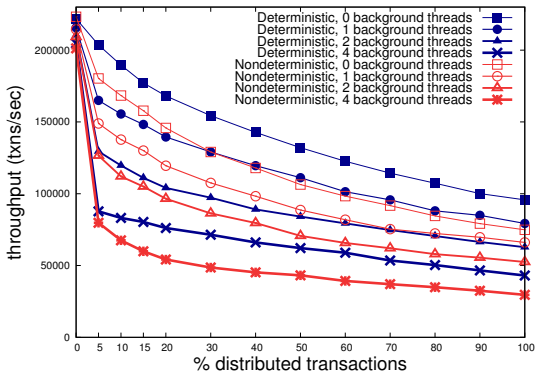
Figure 4: TPC-C throughput.

situations in which one or more servers might start running with suddenly reduced performance.

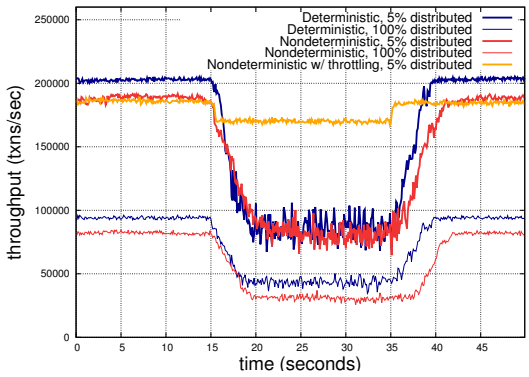
To implement this, we modified one machine out of the eight machines in the cluster in order to slow it down. Specifically, we introduced a number of additional threads in the Calvin process that ran continuous CPU-intensive tasks on the same set of cores that were otherwise reserved for the lock manager and worker threads.

Figure 5(a) illustrates how transactional throughput is affected when one node in the cluster has increasingly limited CPU resources. The biggest difference between these plots and the plots of the previous sections is the extremely sharp drop that appears between 0% and 5% multi-partition transactions when one machine in the cluster is forced to slow down due to resource contention. As described in Section 2, in deterministic systems, it is often impossible for machines to get very far ahead of the slowest machine, since new transactions may have data dependencies on previous ones that access data on that slow machine and which it has not yet processed. Since deterministic transaction execution disallows on-the-fly transaction reordering, the faster machines have no choice but to wait for the data dependency to be satisfied by the slower machine before proceeding.

With no distributed transactions, the effect of slowing down one machine is relatively minor, because as that one machine lags further behind in processing requests from the input log, none of the other machines ever depend on it.



(a) Throughput with background threads on slow node (Long transactions under contention=0.0001).



(b) Throughput vs. time when disruptive tasks are started on one machine in a cluster.

Figure 5: Heterogenous cluster experiments (one node runs slowly due to concurrent background processes).

The mild performance decline here represents only a change in throughput for the one affected server. However, as soon as there are as few as 5% multipartition transactions, data dependencies on the slow node cause all nodes to slow down. This effect increases with more multi-partition transactions and more background threads on the slow node.

Nondeterministic systems do not have the constraint on transaction reordering, and are actually free to change execution order. Surprisingly, this flexibility yielded very little benefit in practice for our nondeterministic implementation. This is because it does not take long for all threads in the thread pool on any particular node (even when we effectively increase the size of the thread pool to over 500 threads via our variation of the thread-per-worker process model described above) to become dependent on data on the slow node, and once this happens throughput of all nodes proceed at the speed of the slowest node.

However, in addition to being able to reorder transactions, nondeterministic systems are also able to arbitrarily abort inconvenient transactions on the fly. To illustrate the protection afforded by this freedom, we implemented an extended version of the nondeterministic system that was designed to "notice" if one machine started having problems and begin to throttle single-partition transaction requests at that partition by preemptively aborting a random subset of single-partition transactions at that machine. This might look like "partial unavailability" since a specific subset of database clients (i.e. those who happened to be interested in data on that particular machine) would suddenly experience very poor performance. However, *overall* throughput would return to near-normal, since the slow machine could devote a larger percentage of its limited resources to executing the transactions that are critical for other machines.

Figure 5(b) shows throughput over time when disruptive CPU-intensive tasks are started on one machine in a cluster. For the nondeterministic implementation with the throttling optimization, the slow machine quickly 'notices' that it is falling behind and begins to abort an increasing percentage of its single-partition transactions, reaching equilibrium once it aborts 70% of its local transactions. As a result, the slow machine was able to focus most of its reduced resources on processing the distributed transactions that other partitions relied upon to make immediate progress. The slow machine was therefore able to avoid impacting the performance of the other seven machines in the cluster[4]. In contrast, the deterministic implementation and the nondeterministic implementation without the throttling optimization were severely impacted by the slow node[5].

Note that we have discussed workloads so far that consist exclusively of read-modify-write transactions. Real-world applications often contain a mix of transactions and read-only queries. In a replicated system, it would be possible for a deterministic mechanism to approximate the above-described throttling behavior by allowing slow machines to reject read requests, forcing the user to retry at a different replica. However, such a technique has some fundamental drawbacks and limitations compared to the above-described mechanism. First, for write-heavy workloads, redirecting read traffic may not be sufficient to reduce usage at a resource-constrained node enough to allow it to keep up with (unrejectable) write requests. Second, if the constrained resource is not CPU, but rather something that is in the critical path for only for writes (such as write bandwidth on an old and poorly wear-leveled flash-drive), rejecting reads may not help transaction throughput at all. Third, it is common for applications to have very stringent latency requirements (especially tail latencies) for read-only operations, while tolerating greater delays for writes. This makes it more desirable in many circumstances to throttle throughput of read-modify-write transactions rather than read throughput.

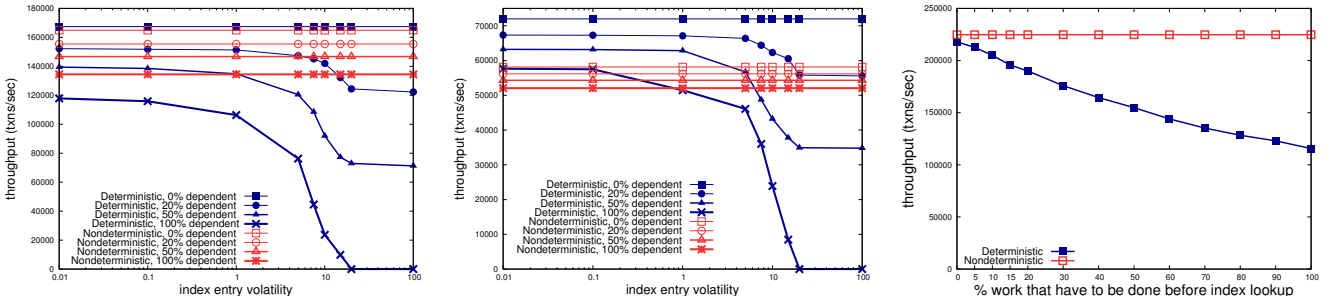
We conclude from these experiments that unpredictable resource constraints represent a significant hazard to deterministic transaction processing systems' performance and reliability.

3.6 Dependent Transactions

As described in Section 2, deterministic locking protocols require a priori knowledge of transactions' read- and write-sets in order to know what locks to request, which requires additional machinery. For some transactions, such as those

[4]We chose to throttle only local transactions because it made this extension very simple to implement and test in the Calvin framework. This technique can also be applied in which the throttled node also aborts some distributed transactions as well, by immediately (or even preemptively) sending abort votes to 2PC coordinators.

[5]In theory, it may be possible to perform throttling in deterministic systems in the preprocessing layer, but such an optimization is beyond the scope of this paper.



(a) %0 distributed transactions (b) %100 distributed transactions (c) Dependent transactions with variable-sized reconnaissance phases

Figure 6: Throughput for transactions in which read/write sets are not known in advance.

that involve secondary index lookups, the determination of the read- and write-sets cannot be done via a static analysis. We refer to such transactions as "dependent transactions".

For such dependent transactions, it is necessary to automatically predict what locks will be requested.One mechanism for doing this is called Optimistic Lock Location Prediction (OLLP), which was described in Section 2. In OLLP, each transaction's execution is broken down into two phases—a "reconnaissance" phase and an "execution" phase. The reconnaissance phase performs any necessary reads, at no isolation, that are needed to determine the transaction's full read-/write-sets. The transaction request is then resubmitted, annotated with the lock location predictions from the reconnaissance phase. These predictions are used by the execution phase to determine what locks to acquire. If these predictions turn out to be incorrect (e.g. due to an update of a secondary index between the reconnaissance and execution phases), the transaction must be (deterministically) aborted and restarted.

To examine the effect of the presence of dependent transactions in a workload, we ran a modified microbenchmark in which certain transactions had to perform a secondary index lockup (which may incur a remote read) in order to determine which 10 records to process. Meanwhile, we concurrently updated the values in that index at various rates. To be as consistent as possible across all measurements, we devoted one core to the thread that updates secondary indexes, and reduced the number of cores available for use by the other execution engine threads from 5 to 4. We use the term "volatility" to refer to the frequency with which each index entry is updated.

In our experiments, we varied the percentage of transactions that were dependent transactions, the percentage of transactions that were distributed, and the index volatility, and we measured both total throughput capacity and latency distributions. We used "long" transactions under "low" contention in order to maximize the detrimental effects of multiple OLLP restarts while minimizing the effects of other purely contention-based performance hazards.

Figure 6(a) shows our measurements when transactions are never distributed. The narrow lines that appear at the top of this figure serve as a baseline and correspond exactly to the left-most points from Figure 1(b)—except that throughput is 20% lower since only 4 cores are allocated to transaction execution, rather than 5. Figure 6(b) shows the same experiment, but with 100% distributed transactions.

In these plots, nondeterministic systems are unaffected

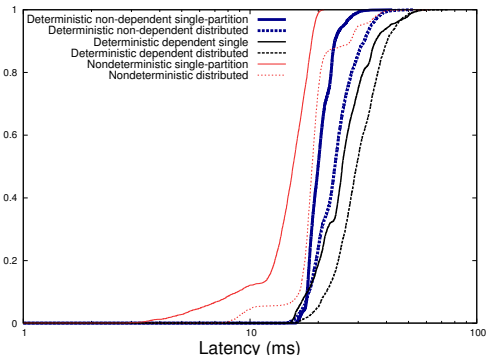


Figure 7: Latency distributions.

by volatility, since they do not require all transactions to predeclare their read- and write-sets, and simply perform the requisite index lookup during execution, before locking each record. However they *are* slightly affected by the percentage of dependent transactions, since in our experimental setup, dependent transactions do more work (an extra index lookup) relative to "ordinary" microbenchmark transactions.

For the deterministic system, when index volatility is low, OLLP requires relatively few transactions to be aborted and restarted. As index volatility rises, increasingly many transactions must be restarted due to inaccurate predictions from the reconnaissance phase, reducing throughput.

Two additional important observations should be made. First, the two plots look very similar. In other words, although the cost of dependent transactions depends heavily on index entry volatility, it does *not* depend on the number of transactions that are distributed. The performance costs of OLLP are therefore *independent* of the performance costs of processing distributed transactions.

Second, index entry volatilities shown here are much higher than those that one might expect to see for secondary indexes in real-world workloads. Recall that an index volatility of 10 does not mean that the secondary index experiences 10 total updates per second—rather, *each entry* in the index is updated 10 times per second. For most real-world volatility levels, OLLP yields very few transaction restarts.

Our previous experiment involved dependent transactions in which the secondary index lockup occurs as the first action in the transaction logic—before any other read/write operations. This limits the cost of the reconnaissance phase

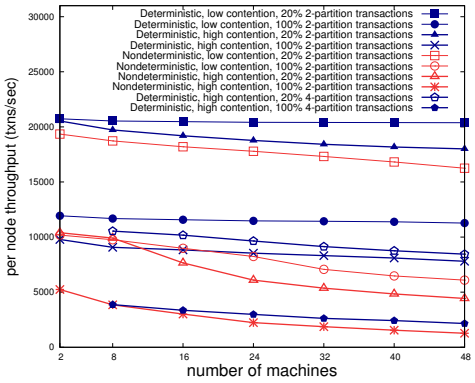


Figure 8: Per-node throughput scalability

in OLLP since it does not need to run the entire transaction — once it performs the index lookup, it can immediate resubmit the transaction since there are no more dependent reads in the transaction after the index lookup. Figure 6(c) shows how performing dependent reads later in a transaction can affect the cost of OLLP. Here, dependent transactions are doing the same total work, but operations are rearranged so that the first X% of the transaction logic does *not* depend on the secondary index lookup and is arranged to occur prior to the lookup.

As Figure 6(c) shows, as the amount of work that needs to be done before all dependencies have been resolved increases, the cost of the reconnaissance step increases and throughput decreases. At the extreme, the reconnaissance step must perform the entire transaction, and therefore overall throughput is halved, since each transaction in-effect must be processed twice (once in reconnaissance and once for actual transaction processing)[6].

3.7 Transaction Latency

Next, we measure execution latencies for the deterministic and nondeterministic execution mechanisms. Figure 7 shows cumulative distribution functions for latencies observed for long transactions with low contention. It is clear that in general, nondeterministic transactions have lower latencies than deterministic transactions. This is because deterministic transactions must go through a preprocessing layer before being processed by the database system. Even with the increased cost of 2PC for distributed transactions, nondeterministic distributed transactions are usually faster than deterministic distributed transactions, because they do not require a preprocessing layer. Dependent transactions further slow down latency for the deterministic database system, because of the reconnaissance phase. Note the log scale used in this plot — there is a often a factor of two difference between nondeterministic latency and deterministic latency.

3.8 Scalability experiments

Finally, we extended several of our measurements to clusters containing up to 48 machines, each containing a partition of the database. Figure 8 shows per-node throughput measurements for low contention ($CI = 0.0001$) and high contention ($CI = 0.01$) workloads. For each contention rate, we measured performance when 20% of transactions were distributed and when *all* transactions were distributed. We find that deterministic execution scales gracefully up to 48 machines, even under high contention. The per-machine throughput decrease as the number of machines grows was due to the phenomenon of *execution progress skew*. Each machine occasionally falls behind briefly in execution due to random variation in workloads, random fluctuations in RPC latencies, etc., slowing down other machines (see Section 3.5); the more machines there are, the more likely that at least one machine is experiencing this at any time. Under higher contention and with more distributed transactions, there is more sensitivity to other machines falling behind and being slow to serve those reads (see Section 3.5), so the effects of execution progress skew are more pronounced in these scenarios, resulting in performance degradation as more machines are added.

Despite the deterministic database being more sensitive to execution progress skew, it still scales better overall than the nondeterministic system. This is because there is an increased number of distributed deadlocks with an increased number of nodes, and as investigated in Section 3.3, distributed deadlocks are a major performance hazard for the nondeterministic system. As contention and/or the percentage of distributed transactions increase, the scalability of the nondeterministic system is increasingly poor.

4. RELATED WORK

Most of the published work on comparisons of deterministic vs. nondeterministic transaction processing have come in the context of the papers introducing new deterministic mechanisms, and therefore focus on workloads that are well-suited for determinism. In contrast, this paper presents a more thorough experimental comparison, examining a wide-range of factors, and focusing on workloads that are both well and poorly-suited for determinism. In particular, the Postgres-R paper experimented with multi-node distributed transaction throughput, transaction latencies, and system scalability [9]. However, they did not vary data contention, size of the transaction, nor number of nodes involved in distributed transactions. Furthermore, they did not look at heterogenous clusters with nodes running unexpectedly slowly. Pacitti et. al. varied transactions size and measured transactions latencies, but did not present any throughput results [17]. Stonebraker et. al. measured throughput on TPC-C, but did not present results on other benchmarks [24]. Therefore, they did not vary data contention, size of the transaction, or percentage of distributed transactions. They also do not experiment with heterogeneous clusters or measure transaction latency. Jimenez-Peris et. al do not present experimental results [7]. Thomson et. al. vary contention, present results on TPC-C, and measure scalability of the system [28]. However, they do not vary the size of the transaction, nor number of nodes involved in distributed transactions. Furthermore, they did not measure performance of dependent transactions where the read/write sets are not known in advance, nor experiment with heterogenous clusters.

Our experiments on dependent transactions relied on the OLLP technique to predict what items a transaction will access (and need to lock) [26]. Predicting access patterns

[6]Our implementation did not contain the optimization that the reconnaissance phase only has to occur at one replica. Had we implemented this, and spread out the reconnaissance work for different transactions across replicas, the overall cost of reconnaissance would be correspondingly smaller.

of transactions in order to optimize processing has been an area of recent focus in the research community, and several promising techniques have emerged, including the use of Markov models [19], static analysis and profiling [2], and run-ahead execution [16].

As mentioned above, there have been many recent proposals for scalable transactional database systems [1, 3, 10, 12, 13, 18, 22, 29, 31]. While these systems present promising approaches to improving nondeterministic transactional throughput, most of the techniques introduced are orthogonal to the question of deterministic vs. nondeterministic transaction processing. Therefore implementing deterministic versions of these systems and comparing them with the original nondeterministic implementations remains an interesting avenue for future work.

5. CONCLUSION

In this paper, we presented an in-depth study that compares deterministic and nondeterministic systems. It is clear that for some workloads deterministic database systems are appropriate, and for some workloads they are not. For latency-sensitive applications, the extra latency required to "preprocess" transactions in deterministic systems can be problematic, and nondeterministic systems may be a better fit. Furthermore, the inability of deterministic database systems to arbitrarily abort transactions to deal with node overload can also limit throughput when there is no other way to reduce the load on a node. However, deterministic database systems are clearly able to scale to higher throughputs of distributed transactions, due largely to their ability to avoid distributed deadlock. Therefore, for workloads that require extreme transactional scalability and do not mind paying the extra latency costs, deterministic database systems are potentially a good fit.

6. ACKNOWLEDGMENTS

This work was sponsored by the NSF under grants IIS-0845643 and IIS-1249722, and by a Sloan Research Fellowship. Kun Ren is also supported by National 973 project under Grant 2012CB316203 and National Natural Science Foundation of China under Grant 61033007.

7. REFERENCES

[1] P. A. Bernstein, C. W. Reid, and S. Das. Hyder - a transactional record manager for shared flash. In *CIDR*, pages 9–20, 2011.

[2] A. Cheung, S. Madden, O. Arden, and A. C. Myers. Speeding up database applications with pyxis. In *SIGMOD*, 2013.

[3] C. Diaconu, C. Freedman, E. Ismert, P. ke Larson, P. Mittal, R. Stonecipher, N. Verma, and M. Zwilling. Hekaton: Sql server's memory-optimized oltp engine. In *SIGMOD*, 2013.

[4] J. Gray. *Notes on database operating systems*. Operating System, An Advanced Course. Springer-Verlag,Berlin, 1979.

[5] S. Harizopoulos, D. J. Abadi, S. R. Madden, and M. Stonebraker. OLTP through the looking glass, and what we found there. In *SIGMOD*, 2008.

[6] J. M. Hellerstein, M. Stonebraker, and J. Hamilton. Architecture of a database system. *Found. Trends databases*, 1(2):141–259, Feb. 2007.

[7] R. Jimenez-Peris, M. Patino-Martinez, and S. Arevalo. Deterministic scheduling for transactional multithreaded replicas. In *IEEE SRDS*, 2000.

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

[9] B. Kemme and G. Alonso. Don't be lazy, be consistent: Postgres-r, a new way to implement database replication. In *VLDB*, 2000.

[10] A. Kemper, T. Neumann, J. Finis, F. Funke, V. Leis, H. Muhe, T. Muhlbauer, and W. Rodiger. Transaction Processing in the Hybrid OLTP/OLAP Main-Memory Database System HyPer. *In IEEE Data Engineering Bulletin*, June 2013.

[11] E. Koskinen and M. Herlihy. Dreadlocks: Efficient deadlock detection. In *Proc. of SPAA*, pages 297–303, 2008.

[12] P. Larson, S. Blanas, C. Diaconu, C. Freedman, J. Patel, and M. Zwilling. High-performance concurrency control mechanisms for main-memory database. *PVLDB*, 2011.

[13] J. Lee, M. Muehle, N. May, F. Faerber, V. Sikka1, H. Plattner, J. Krueger, and M. Grund. High-performance transaction processing in sap hana. *In IEEE Data Engineering Bulletin.*, June 2013.

[14] N. Malviya. *Recovery Algorithms for In-Memory OLTP Databases*. MS thesis, MIT, 2012.

[15] N. Malviya, A. Weisberg, S. Madden, and M. Stonebraker. Rethinking main memory oltp recovery. In *Proc. of ICDE*, 2014.

[16] O. Mutlu, H. Kim, and Y. N. Patt. Techniques for efficient processing in runahead execution engines. In *ISCA*, 2005.

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

[18] I. Pandis, R. Johnson, N. Hardavellas, and A. Ailamaki. Data-oriented transaction execution. *PVLDB*, 3(1):928–939, 2010.

[19] A. Pavlo, E. P. Jones, and S. Zdonik. On predictive modeling for optimizing transaction execution in parallel oltp systems. *PVLDB*, 5(2):85–96, 2012.

[20] K. Ren, A. Thomson, and D. J. Abadi. Lightweight locking for main memory database systems. *PVLDB*, 2013.

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

[22] J. Shute, R. Vingralek, B. Samwel, B. Handy, C. Whipkey, E. Rollins, M. Oancea, K. Littleeld, D. Menestrina, S. Ellner, J. Cieslewicz, I. Rae, T. Stancescu, and H. Apte. F1: A distributed sql database that scales. In *VLDB*, 2013.

[23] M. Stonebraker. Concurrency control and consistency of multiple copies of data in distributed ingres. *IEEE Transactions on Software Engineering*, SE-5, 1979.

[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 *VLDB*, Vienna, Austria, 2007.

[25] M. Stonebraker and A. Weisberg. The voltdb main memory dbms. *In IEEE Data Engineering Bulletin.*, June 2013.

[26] A. Thomson and D. J. Abadi. The case for determinism in database systems. *VLDB*, 2010.

[27] A. Thomson and D. J. Abadi. Modularity and scalability in calvin. *In IEEE Data Engineering Bulletin.*, June 2013.

[28] A. Thomson, T. Diamond, P. Shao, K. Ren, S.-C. Weng, and D. J. Abadi. Calvin: Fast distributed transactions for partitioned database systems. In *SIGMOD*, 2012.

[29] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden. Speedy transactions in multicore in-memory databases. In *Proc. of SOSP*, SOSP '13, pages 18–32, 2013.

[30] A. Weisberg. Voltdb 3.x features and performance. A few more amps. hemptember 30 2012. ttp://www.afewmoreamps.com/2012/09/voltdb-3x-features-and-performance.html.

[31] J. L. om, V. Raatikka, J. Ruuth, P. Soini, and K. Vakkila. Ibm soliddb: In-memory database optimized for extreme speed and availability. *In IEEE Data Engineering Bulletin.*, June 2013.