Calvin: Fast Distributed Transactions for Partitioned Database Systems

Alexander Thomson
Yale University
thomson@cs.yale.edu

Thaddeus Diamond
Yale University
diamond@cs.yale.edu

Shu-Chun Weng
Yale University
scweng@cs.yale.edu

Kun Ren
Yale University
kun@cs.yale.edu

Philip Shao
Yale University
shao-philip@cs.yale.edu

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

ABSTRACT

Many distributed storage systems achieve high data access throughput via partitioning and replication, each system with its own advantages and tradeoffs. In order to achieve high scalability, however, today's systems generally reduce transactional support, disallowing single transactions from spanning multiple partitions. Calvin is a practical transaction scheduling and data replication layer that uses a deterministic ordering guarantee to significantly reduce the normally prohibitive contention costs associated with distributed transactions. Unlike previous deterministic database system prototypes, Calvin supports disk-based storage, scales near-linearly on a cluster of commodity machines, and has no single point of failure. By replicating transaction inputs rather than effects, Calvin is also able to support multiple consistency levels—including Paxos-based strong consistency across geographically distant replicas—at no cost to transactional throughput.

Categories and Subject Descriptors

C.2.4 [**Distributed Systems**]: Distributed databases;
H.2.4 [**Database Management**]: Systems—*concurrency, distributed databases, transaction processing*

General Terms

Algorithms, Design, Performance, Reliability

Keywords

determinism, distributed database systems, replication, transaction processing

SIGMOD '12, May 20–24, 2012, Scottsdale, Arizona, USA.

1. BACKGROUND AND INTRODUCTION

One of several current trends in distributed database system design is a move away from supporting traditional ACID database transactions. Some systems, such as Amazon's Dynamo [13], MongoDB [24], CouchDB [6], and Cassandra [17] provide no transactional support whatsoever. Others provide only limited transactionality, such as single-row transactional updates (e.g. Bigtable [11]) or transactions whose accesses are limited to small subsets of a database (e.g. Azure [9], Megastore [7], and the Oracle NoSQL Database [26]). The primary reason that each of these systems does not support fully ACID transactions is to provide linear outward scalability. Other systems (e.g. VoltDB [27, 16]) support full ACID, but cease (or limit) concurrent transaction execution when processing a transaction that accesses data spanning multiple partitions.

Reducing transactional support greatly simplifies the task of building linearly scalable distributed storage solutions that are designed to serve "embarrassingly partitionable" applications. For applications that are not easily partitionable, however, the burden of ensuring atomicity and isolation is generally left to the application programmer, resulting in increased code complexity, slower application development, and low-performance client-side transaction scheduling.

Calvin is designed to run alongside a non-transactional storage system, transforming it into a shared-nothing (near-)linearly scalable database system that provides high availability[1] and full ACID transactions. These transactions can potentially span multiple partitions spread across the shared-nothing cluster. Calvin accomplishes this by providing a layer above the storage system that handles the scheduling of distributed transactions, as well as replication and network communication in the system. The key technical feature that allows for scalability in the face of distributed transactions is a deterministic locking mechanism that enables the elimination of distributed commit protocols.

[1]In this paper we use the term "high availability" in the common colloquial sense found in the database community where a database is *highly available* if it can fail over to an active replica on the fly with no downtime, rather than the definition of high availability used in the CAP theorem which requires that even minority replicas remain available during a network partition.

1.1 The cost of distributed transactions

Distributed transactions have historically been implemented by the database community in the manner pioneered by the architects of System R* [22] in the 1980s. The primary mechanism by which System R*-style distributed transactions impede throughput and extend latency is the requirement of an agreement protocol between all participating machines at commit time to ensure atomicity and durability. To ensure isolation, all of a transaction's locks must be held for the full duration of this agreement protocol, which is typically two-phase commit.

The problem with holding locks during the agreement protocol is that two-phase commit requires multiple network round-trips between all participating machines, and therefore the time required to run the protocol can often be considerably greater than the time required to execute all local transaction logic. If a few popularly-accessed records are frequently involved in distributed transactions, the resulting extra time that locks are held on these records can have an extremely deleterious effect on overall transactional throughput. We refer to the total duration that a transaction holds its locks—which includes the duration of any required commit protocol—as the transaction's *contention footprint*. Although most of the discussion in this paper assumes pessimistic concurrency control mechanisms, the costs of extending a transaction's contention footprint are equally applicable—and often even worse due to the possibility of cascading aborts—in optimistic schemes.

Certain optimizations to two-phase commit, such as combining multiple concurrent transactions' commit decisions into a single round of the protocol, can reduce the CPU and network overhead of two-phase commit, but do not ameliorate its contention cost.

Allowing distributed transactions may also introduce the possibility of distributed deadlock in systems implementing pessimistic concurrency control schemes. While detecting and correcting deadlocks does not typically incur prohibitive system overhead, it can cause transactions to be aborted and restarted, increasing latency and reducing throughput to some extent.

1.2 Consistent replication

A second trend in distributed database system design has been towards reduced consistency guarantees with respect to replication. Systems such as Dynamo, SimpleDB, Cassandra, Voldemort, Riak, and PNUTS all lessen the consistency guarantees for replicated data [13, 1, 17, 2, 3, 12]. The typical reason given for reducing the replication consistency of these systems is the CAP theorem [5, 14]—in order for the system to achieve 24/7 global availability and remain available even in the event of a network partition, the system must provide lower consistency guarantees. However, in the last year, this trend is starting to reverse—perhaps in part due to ever-improving global information infrastructure that makes non-trivial network partitions increasingly rare—with several new systems supporting strongly consistent replication. Google's Megastore [7] and IBM's Spinnaker [25], for example, are synchronously replicated via Paxos [18, 19].

Synchronous updates come with a latency cost fundamental to the agreement protocol, which is dependent on network latency between replicas. This cost can be significant, since replicas are often geographically separated to reduce correlated failures. However, this is intrinsically a latency cost only, and need not necessarily affect contention or throughput.

1.3 Achieving agreement without increasing contention

Calvin's approach to achieving inexpensive distributed transactions and synchronous replication is the following: when multiple machines need to agree on how to handle a particular transaction, they do it outside of transactional boundaries—that is, *before* they acquire locks and begin executing the transaction.

Once an agreement about how to handle the transaction has been reached, it must be executed to completion according to the plan—node failure and related problems cannot cause the transaction to abort. If a node fails, it can recover from a replica that had been executing the same plan in parallel, or alternatively, it can replay the history of planned activity for that node. Both parallel plan execution and replay of plan history require activity plans to be deterministic—otherwise replicas might diverge or history might be repeated incorrectly.

To support this determinism guarantee while maximizing concurrency in transaction execution, Calvin uses a deterministic locking protocol based on one we introduced in previous work [28].

Since all Calvin nodes reach an agreement regarding what transactions to attempt and in what order, it is able to completely eschew distributed commit protocols, reducing the contention footprints of distributed transactions, thereby allowing throughput to scale out nearly linearly despite the presence of multipartition transactions. Our experiments show that Calvin significantly outperforms traditional distributed database designs under high contention workloads. We find that it is possible to run half a million TPC-C transactions per second on a cluster of commodity machines in the Amazon cloud, which is immediately competitive with the world-record results currently published on the TPC-C website that were obtained on much higher-end hardware.

This paper's primary contributions are the following:

- The design of a transaction scheduling and data replication layer that transforms a non-transactional storage system into a (near-)linearly scalable shared-nothing database system that provides high availability, strong consistency, and full ACID transactions.
- A practical implementation of a deterministic concurrency control protocol that is more scalable than previous approaches, and does not introduce a potential single point of failure.
- A data prefetching mechanism that leverages the planning phase performed prior to transaction execution to allow transactions to operate on disk-resident data without extending transactions' contention footprints for the full duration of disk lookups.
- A fast checkpointing scheme that, together with Calvin's determinism guarantee, completely removes the need for physical REDO logging and its associated overhead.

The following section discusses further background on deterministic database systems. In Section 3 we present Calvin's architecture. In Section 4 we address how Calvin handles transactions that access disk-resident data. Section 5 covers Calvin's mechanism for periodically taking full database snapshots. In Section 6 we present a series of experiments that explore the throughput and latency of Calvin under different workloads. We present related work in Section 7, discuss future work in Section 8, and conclude in Section 9.

2. DETERMINISTIC DATABASE SYSTEMS

In traditional (System R*-style) distributed database systems, the primary reason that an agreement protocol is needed when committing a distributed transaction is to ensure that all effects of a transaction have successfully made it to durable storage in an atomic

fashion—either all nodes involved the transaction agree to "commit" their local changes or none of them do. Events that prevent a node from committing its local changes (and therefore cause the entire transaction to abort) fall into two categories: nondeterministic events (such as node failures) and deterministic events (such as transaction logic that forces an abort if, say, an inventory stock level would fall below zero otherwise).

There is no fundamental reason that a transaction must abort as a result of any nondeterministic event; when systems do choose to abort transactions due to outside events, it is due to practical consideration. After all, forcing all other nodes in a system to wait for the node that experienced a nondeterministic event (such as a hardware failure) to recover could bring a system to a painfully long stand-still.

If there is a replica node performing the exact same operations in parallel to a failed node, however, then other nodes that depend on communication with the afflicted node to execute a transaction need not wait for the failed node to recover back to its original state—rather they can make requests to the replica node for any data needed for the current or future transactions. Furthermore, the transaction can be committed since the replica node was able to complete the transaction, and the failed node will eventually be able to complete the transaction upon recovery[2].

Therefore, if there exists a replica that is processing the same transactions in parallel to the node that experiences the nondeterministic failure, the requirement to abort transactions upon such failures is eliminated. The only problem is that replicas need to be going through the same sequence of database states in order for a replica to immediately replace a failed node in the middle of a transaction. Synchronously replicating every database state change would have far too high of an overhead to be feasible. Instead, deterministic database systems synchronously replicate batches of transaction *requests*. In a traditional database implementation, simply replicating transactional input is not generally sufficient to ensure that replicas do not diverge, since databases guarantee that they will process transactions in a manner that is logically equivalent to *some* serial ordering of transactional input—but two replicas may choose to process the input in manners equivalent to different serial orders, for example due to different thread scheduling, network latencies, or other hardware constraints. However, if the concurrency control layer of the database is modified to acquire locks in the order of the agreed upon transactional input (and several other minor modifications to the database are made [28]), all replicas can be made to emulate the same serial execution order, and database state can be guaranteed not to diverge[3].

Such deterministic databases allow two replicas to stay consistent simply by replicating database input, and as described above, the presence of these actively replicated nodes enable distributed transactions to commit their work in the presence of nondeterministic failures (which can potentially occur in the middle of a transaction). This eliminates the primary justification for an agreement protocol at the end of distributed transactions (the need to check for a node failure which could cause the transaction to abort). The other potential cause of an abort mentioned above—deterministic logic in the transaction (e.g. a transaction should be aborted if inventory is zero)—does not necessarily have to be performed as part of an agreement protocol at the end of a transaction. Rather, each node involved in a transaction waits for a one-way message from each node that could potentially deterministically abort the transaction, and only commits once it receives these messages.

[2]Even in the unlikely event that all replicas experience the same nondeterministic failure, the transaction can still be committed if there was no deterministic code in the part of the transaction assigned to the failed nodes that could cause the transaction to abort.

[3]More precisely, the replica states are guaranteed not to appear divergent to outside requests for data, even though their physical states are typically not identical at any particular snapshot of the system.

3. SYSTEM ARCHITECTURE

Calvin is designed to serve as a scalable transactional layer above any storage system that implements a basic CRUD interface (create/insert, read, update, and delete). Although it is possible to run Calvin on top of distributed non-transactional storage systems such as SimpleDB or Cassandra, it is more straightforward to explain the architecture of Calvin assuming that the storage system is not distributed out of the box. For example, the storage system could be a single-node key-value store that is installed on multiple independent machines ("nodes"). In this configuration, Calvin organizes the partitioning of data across the storage systems on each node, and orchestrates all network communication that must occur between nodes in the course of transaction execution.

The high level architecture of Calvin is presented in Figure 1. The essence of Calvin lies in separating the system into three separate layers of processing:

- The **sequencing layer** (or "sequencer") intercepts transactional inputs and places them into a global transactional input sequence—this sequence will be the order of transactions to which all replicas will ensure serial equivalence during their execution. The sequencer therefore also handles the replication and logging of this input sequence.

- The **scheduling layer** (or "scheduler") orchestrates transaction execution using a deterministic locking scheme to guarantee equivalence to the serial order specified by the sequencing layer while allowing transactions to be executed concurrently by a pool of transaction execution threads. (Although they are shown below the scheduler components in Figure 1, these execution threads conceptually belong to the scheduling layer.)

- The **storage layer** handles all physical data layout. Calvin transactions access data using a simple CRUD interface; any storage engine supporting a similar interface can be plugged into Calvin fairly easily.

All three layers scale horizontally, their functionalities partitioned across a cluster of shared-nothing nodes. Each node in a Calvin deployment typically runs one partition of each layer (the tall light-gray boxes in Figure 1 represent physical machines in the cluster). We discuss the implementation of these three layers in the following sections.

By separating the replication mechanism, transactional functionality and concurrency control (in the sequencing and scheduling layers) from the storage system, the design of Calvin deviates significantly from traditional database design which is highly monolithic, with physical access methods, buffer manager, lock manager, and log manager highly integrated and cross-reliant. This decoupling makes it impossible to implement certain popular recovery and concurrency control techniques such as the physiological logging in ARIES and next-key locking technique to handle phantoms (i.e., using physical surrogates for logical properties in concurrency control). Calvin is not the only attempt to separate the transactional components of a database system from the data components—thanks to cloud computing and its highly modular

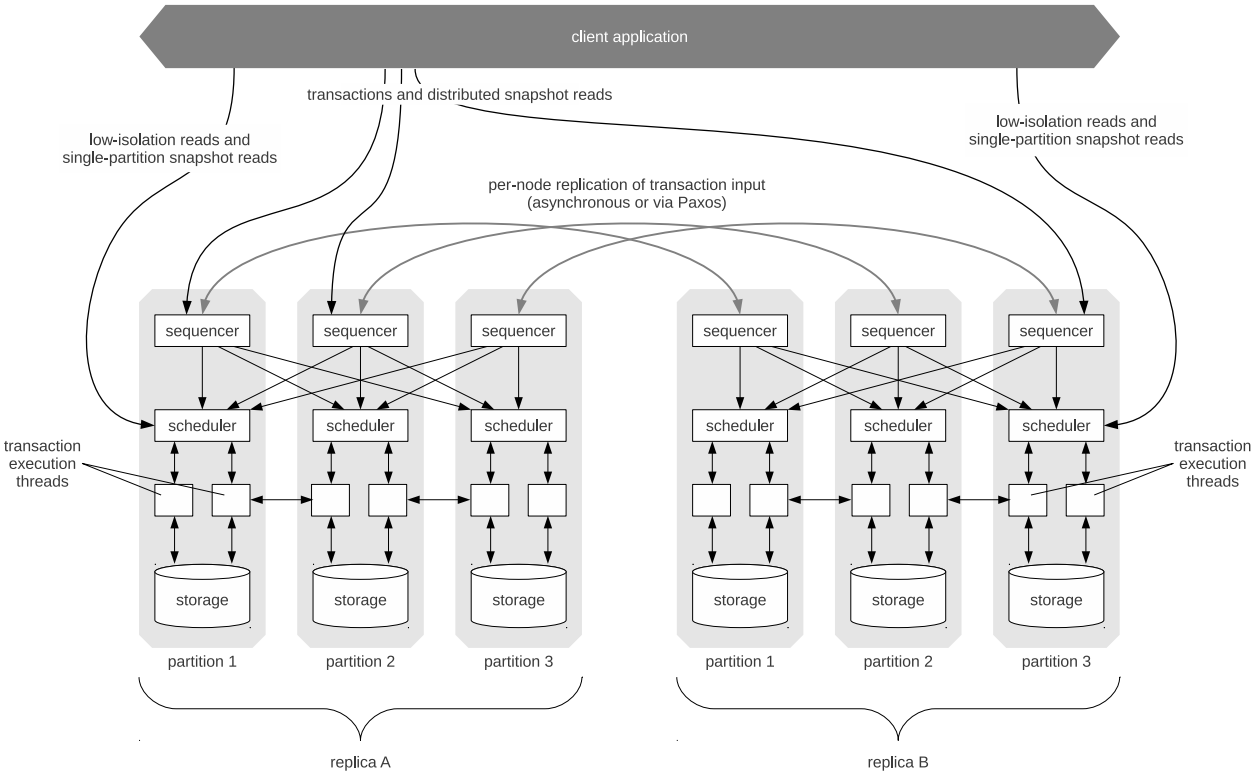

Figure 1: System Architecture of Calvin

services, there has been a renewed interest within the database community in separating these functionalities into distinct and modular system components [21].

3.1 Sequencer and replication

In previous work with deterministic database systems, we implemented the sequencing layer's functionality as a simple echo server—a single node which accepted transaction requests, logged them to disk, and forwarded them in timestamp order to the appropriate database nodes within each replica [28]. The problems with single-node sequencers are (a) that they represent potential single points of failure and (b) that as systems grow the constant throughput bound of a single-node sequencer brings overall system scalability to a quick halt. Calvin's sequencing layer is distributed across all system replicas, and also partitioned across every machine within each replica.

Calvin divides time into 10-millisecond epochs during which every machine's sequencer component collects transaction requests from clients. At the end of each epoch, all requests that have arrived at a sequencer node are compiled into a batch. This is the point at which replication of transactional inputs (discussed below) occurs.

After a sequencer's batch is successfully replicated, it sends a message to the scheduler on every partition within its replica containing (1) the sequencer's unique node ID, (2) the epoch number (which is synchronously incremented across the entire system once every 10 ms), and (3) all transaction inputs collected that the recipient will need to participate in. This allows every scheduler to piece together its own view of a global transaction order by interleaving (in a deterministic, round-robin manner) all sequencers' batches for that epoch.

3.1.1 *Synchronous and asynchronous replication*

Calvin currently supports two modes for replicating transactional input: asynchronous replication and Paxos-based synchronous replication. In both modes, nodes are organized into *replication groups*, each of which contains all replicas of a particular partition. In the deployment in Figure 1, for example, partition 1 in replica A and partition 1 in replica B would together form one replication group.

In asynchronous replication mode, one replica is designated as a master replica, and all transaction requests are forwarded immediately to sequencers located at nodes of this replica. After compiling each batch, the sequencer component on each master node forwards the batch to all other (slave) sequencers in its replication group. This has the advantage of extremely low latency before a transaction can begin being executed at the master replica, at the cost of significant complexity in failover. On the failure of a master sequencer, agreement has to be reached between all nodes in the same replica *and* all members of the failed node's replication group regarding (a) which batch was the last valid batch sent out by the failed sequencer and (b) exactly what transactions that batch contained, since each scheduler is only sent the partial view of each batch that it actually needs in order to execute.

Calvin also supports Paxos-based synchronous replication of transactional inputs. In this mode, all sequencers within a replication group use Paxos to agree on a combined batch of transaction requests for each epoch. Calvin's current implementation uses ZooKeeper, a highly reliable distributed coordination service often used by distributed database systems for heartbeats, configuration syn-

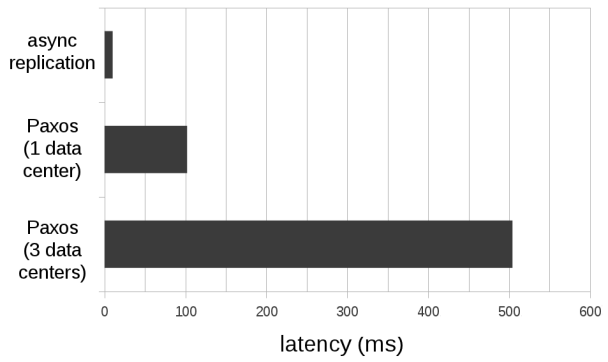

Figure 2: Average transaction latency under Calvin's different replication modes.

chronization and naming [15]. ZooKeeper is not optimized for storing high data volumes, and may incur higher total latencies than the most efficient possible Paxos implementations. However, ZooKeeper handles the necessary *throughput* to replicate Calvin's transactional inputs for all the experiments run in this paper, and since this synchronization step does not extend contention footprints, transactional throughput is completely unaffected by this preprocessing step. Improving the Calvin codebase by implementing a more streamlined Paxos agreement protocol between Calvin sequencers than what comes out-of-the-box with ZooKeeper could be useful for latency-sensitive applications, but would not improve Calvin's transactional throughput.

Figure 2 presents average transaction latencies for the current Calvin codebase under different replication modes. The above data was collected using 4 EC2 High-CPU machines per replica, running 40000 microbenchmark transactions per second (10000 per node), 10% of which were multipartition (see Section 6 for additional details on our experimental setup). Both Paxos latencies reported used three replicas (12 total nodes). When all replicas were run on one data center, ping time between replicas was approximately 1ms. When replicating across data centers, one replica was run on Amazon's East US (Virginia) data center, one was run on Amazon's West US (Northern California) data center, and one was run on Amazon's EU (Ireland) data center. Ping times between replicas ranged from 100 ms to 170 ms. Total transactional throughput was *not* affected by changing Calvin's replication mode.

3.2 Scheduler and concurrency control

When the transactional component of a database system is unbundled from the storage component, it can no longer make any assumptions about the physical implementation of the data layer, and cannot refer to physical data structures like pages and indexes, nor can it be aware of side-effects of a transaction on the physical layout of the data in the database. Both the logging and concurrency protocols have to be completely logical, referring only to record keys rather than physical data structures. Fortunately, the inability to perform physiological logging is not at all a problem in deterministic database systems; since the state of a database can be completely determined from the input to the database, logical logging is straightforward (the input is be logged by the sequencing layer, and occasional checkpoints are taken by the storage layer—see Section 5 for further discussion of checkpointing in Calvin).

However, only having access to logical records is slightly more problematic for concurrency control, since locking ranges of keys and being robust to phantom updates typically require physical access to the data. To handle this case, Calvin could use an approach proposed recently for another unbundled database system by creating virtual resources that can be logically locked in the transactional layer [20], although implementation of this feature remains future work.

Calvin's deterministic lock manager is partitioned across the entire scheduling layer, and each node's scheduler is only responsible for locking records that are stored at that node's storage component—even for transactions that access records stored on other nodes. The locking protocol resembles strict two-phase locking, but with two added invariants:

- For any pair of transactions A and B that both request exclusive locks on some local record R, if transaction A appears before B in the serial order provided by the sequencing layer then A must request its lock on R before B does. In practice, Calvin implements this by serializing all lock requests in a single thread. The thread scans the serial transaction order sent by the sequencing layer; for each entry, it requests all locks that the transaction will need in its lifetime. (All transactions are therefore required to declare their full read/write sets in advance; section 3.2.1 discusses the limitations entailed.)

- The lock manager must grant each lock to requesting transactions strictly in the order in which those transactions requested the lock. So in the above example, B could not be granted its lock on R until after A has acquired the lock on R, executed to completion, and released the lock.

Clients specify transaction logic as C++ functions that may access any data using a basic CRUD interface. Transaction code does not need to be at all aware of partitioning (although the user may specify elsewhere how keys should be partitioned across machines), since Calvin intercepts all data accesses that appear in transaction code and performs all remote read result forwarding automatically.

Once a transaction has acquired all of its locks under this protocol (and can therefore be safely executed in its entirety) it is handed off to a worker thread to be executed. Each actual transaction execution by a worker thread proceeds in five phases:

1. **Read/write set analysis.** The first thing a transaction execution thread does when handed a transaction request is analyze the transaction's read and write sets, noting (a) the elements of the read and write sets that are stored locally (i.e. at the node on which the thread is executing), and (b) the set of participating nodes at which elements of the write set are stored. These nodes are called *active participants* in the transaction; participating nodes at which only elements of the read set are stored are called *passive participants*.

2. **Perform local reads.** Next, the worker thread looks up the values of all records in the read set that are stored locally. Depending on the storage interface, this may mean making a copy of the record to a local buffer, or just saving a pointer to the location in memory at which the record can be found.

3. **Serve remote reads.** All results from the local read phase are forwarded to counterpart worker threads on every *actively* participating node. Since passive participants do not modify any data, they need not execute the actual transaction code, and therefore do not have to collect any remote read results.

If the worker thread is executing at a passively participating node, then it is finished after this phase.

4. **Collect remote read results.** If the worker thread is executing at an actively participating node, then it must execute transaction code, and thus it must first acquire all read results—both the results of local reads (acquired in the second phase) and the results of remote reads (forwarded appropriately by every participating node during the third phase). In this phase, the worker thread collects the latter set of read results.

5. **Transaction logic execution and applying writes.** Once the worker thread has collected all read results, it proceeds to execute all transaction logic, applying any *local* writes. Non-local writes can be ignored, since they will be viewed as local writes by the counterpart transaction execution thread at the appropriate node, and applied there.

Assuming a distributed transaction begins executing at approximately the same time at every participating node (which is not always the case—this is discussed in greater length in Section 6), all reads occur in parallel, and all remote read results are delivered in parallel as well, with no need for worker threads at different nodes to request data from one another at transaction execution time.

3.2.1 Dependent transactions

Transactions which must perform reads in order to determine their full read/write sets (which we term *dependent transactions*) are not natively supported in Calvin since Calvin's deterministic locking protocol requires advance knowledge of all transactions' read/write sets before transaction execution can begin. Instead, Calvin supports a scheme called Optimistic Lock Location Prediction (OLLP), which can be implemented at very low overhead cost by modifying the client transaction code itself [28]. The idea is for dependent transactions to be preceded by an inexpensive, low-isolation, unreplicated, read-only *reconnaissance query* that performs all the necessary reads to discover the transaction's full read/write set. The *actual* transaction is then sent to be added to the global sequence and executed, using the reconnaissance query's results for its read/write set. Because it is possible for the records read by the reconnaissance query (and therefore the actual transaction's read/write set) to have changed between the execution of the reconnaissance query and the execution of the actual transaction, the read results must be rechecked, and the process have to may be (deterministically) restarted if the "reconnoitered" read/write set is no longer valid.

Particularly common within this class of transactions are those that must perform secondary index lookups in order to identify their full read/write sets. Since secondary indexes tend to be comparatively expensive to modify, they are seldom kept on fields whose values are updated extremely frequently. Secondary indexes on "inventory item *name*" or "New York Stock Exchange stock *symbol*", for example, would be common, whereas it would be unusual to maintain a secondary index on more volatile fields such as "inventory item *quantity*" or "NYSE stock *price*". One therefore expects the OLLP scheme seldom to result in repeated transaction restarts under most common real-world workloads.

The TPC-C benchmark's "Payment" transaction type is an example of this sub-class of transaction. And since the TPC-C benchmark workload *never* modifies the index on which Payment transactions' read/write sets may depend, Payment transactions never have to be restarted when using OLLP.

4. CALVIN WITH DISK-BASED STORAGE

Our previous work on deterministic database system came with the caveat that deterministic execution would only work for databases entirely resident in main memory [28]. The reasoning was that a major disadvantage of deterministic database systems relative to traditional nondeterministic systems is that nondeterministic systems are able to guarantee equivalence to *any* serial order, and can therefore arbitrarily reorder transactions, whereas a system like Calvin is constrained to respect whatever order the sequencer chooses.

For example, if a transaction (let's call it A) is stalled waiting for a disk access, a traditional system would be able to run other transactions (B and C, say) that do not conflict with the locks already held by A. If B and C's write sets overlapped with A's on keys that A has not yet locked, then execution can proceed in manner equivalent to the serial order $B - C - A$ rather than $A - B - C$. In a deterministic system, however, B and C would have to block until A completed. Worse yet, other transactions that conflicted with B and C—but *not* with A—would also get stuck behind A. On-the-fly reordering is therefore highly effective at maximizing resource utilization in systems where disk stalls upwards of 10 ms may occur frequently during transaction execution.

Calvin avoids this disadvantage of determinism in the context of disk-based databases by following its guiding design principle: move as much as possible of the heavy lifting to *earlier* in the transaction processing pipeline, *before* locks are acquired.

Any time a sequencer component receives a request for a transaction that may incur a disk stall, it introduces an artificial delay before forwarding the transaction request to the scheduling layer and meanwhile sends requests to all relevant storage components to "warm up" the disk-resident records that the transaction will access. If the artificial delay is greater than or equal to the time it takes to bring all the disk-resident records into memory, then when the transaction is actually executed, it will access only memory-resident data. Note that with this scheme the overall latency for the transaction should be no greater than it would be in a traditional system where the disk IO were performed during execution (since exactly the same set of disk operations occur in either case)—but none of the disk latency adds to the transaction's contention footprint.

To clearly demonstrate the applicability (and pitfalls) of this technique, we implemented a simple disk-based storage system for Calvin in which "cold" records are written out to the local filesystem and only read into Calvin's primary memory-resident key-value table when needed by a transaction. When running 10,000 microbenchmark transactions per second per machine (see Section 6 for more details on experimental setup), Calvin's total transactional throughput was unaffected by the presence of transactions that access disk-based storage, as long as no more than 0.9% of transactions (90 out of 10,000) to disk. However, this number is very dependent on the particular hardware configuration of the servers used. We ran our experiments on low-end commodity hardware, and so we found that the number of disk-accessing transactions that could be supported was limited by the maximum *throughput* of local disk (rather than contention footprint). Since the microbenchmark workload involved random accesses to a lot of different files, 90 disk-accessing transactions per second per machine was sufficient to turn disk random access throughput into a bottleneck. With higher end disk arrays (or with flash memory instead of magnetic disk) many more disk-based transactions could be supported without affecting total throughput in Calvin.

To better understand Calvin's potential for interfacing with other disk configurations, flash, networked block storage, etc., we also implemented a storage engine in which "cold" data was stored in

memory on a separate machine that could be configured to serve data requests only after a pre-specified delay (to simulate network or storage-access latency). Using this setup, we found that each machine was able to support the same load of 10,000 transactions per second, no matter how many of these transactions accessed "cold" data—even under extremely high contention (contention index = 0.01).

We found two main challenges in reconciling deterministic execution with disk-based storage. First, disk latencies must be accurately predicted so that transactions are delayed for the appropriate amount of time. Second, Calvin's sequencer layer must accurately track which keys are in memory across all storage nodes in order to determine when prefetching is necessary.

4.1 Disk I/O latency prediction

Accurately predicting the time required to fetch a record from disk to memory is not an easy problem. The time it takes to read a disk-resident can vary significantly for many reasons:

- Variable physical distance for the head and spindle to move
- Prior queued disk I/O operations
- Network latency for remote reads
- Failover from media failures
- Multiple I/O operations required due to traversing a disk-based data structure (e.g. a B+ tree)

It is therefore impossible to predict latency perfectly, and any heuristic used will sometimes result in underestimates and sometimes in overestimates. Disk IO latency estimation proved to be a particularly interesting and crucial parameter when tuning Calvin to perform well on disk-resident data under high contention.

We found that if the sequencer chooses a conservatively high estimate and delays forwarding transactions for longer than is likely necessary, the contention cost due to disk access is minimized (since fetching is almost always completed before the transaction requires the record to be read), but at a cost to overall transaction latency. Excessively high estimates could also result in the memory of the storage system being overloaded with "cold" records waiting for the transactions that requested them to be scheduled.

However, if the sequencer underestimates disk I/O latency and does not delay the transaction for long enough, then it will be scheduled too soon and stall during execution until all fetching completes. Since locks are held for the duration, this may come with high costs to contention footprint and therefore overall throughput.

There is therefore a fundamental tradeoff between total transactional latency and contention when estimating for disk I/O latency. In both experiments described above, we tuned our latency predictions so at least 99% of disk-accessing transactions were scheduled *after* their corresponding prefetching requests had completed. Using the simple filesystem-based storage engine, this meant introducing an artificial delay of 40ms, but this was sufficient to sustain throughput even under very high contention (contention index = 0.01). Under lower contention (contention index $\leq$ 0.001), we found that no delay was necessary beyond the default delay caused by collecting transaction requests into batches, which averages 5 ms. A more exhaustive exploration of this particular latency-contention tradeoff would be an interesting avenue for future research, particularly as we experiment further with hooking Calvin up to various commercially available storage engines.

4.2 Globally tracking hot records

In order for the sequencer to accurately determine which transactions to delay scheduling while their read sets are warmed up, each node's sequencer component must track what data is currently in memory across the entire system—not just the data managed by the storage components co-located on the sequencer's node. Although this was feasible for our experiments in this paper, this is not a scalable solution. If global lists of hot keys are not tracked at every sequencer, one solution is to delay *all* transactions from being scheduled until adequate time for prefetching has been allowed. This protects against disk seeks extending contention footprints, but incurs latency at every transaction. Another solution (for single-partition transactions only) would be for schedulers to track their local hot data synchronously across all replicas, and then allow schedulers to deterministically decide to delay requesting locks for single-partition transactions that try to read cold data. A more comprehensive exploration of this strategy, including investigation of how to implement it for multipartition transactions, remains future work.

5. CHECKPOINTING

Deterministic database systems have two properties that simplify the task of ensuring fault tolerance. First, active replication allows clients to instantaneously failover to another replica in the event of a crash.

Second, only the transactional input is logged—there is no need to pay the overhead of physical REDO logging. Replaying history of transactional input is sufficient to recover the database system to the current state. However, it would be inefficient (and ridiculous) to replay the entire history of the database from the beginning of time upon every failure. Instead, Calvin periodically takes a checkpoint of full database state in order to provide a starting point from which to begin replay during recovery.

Calvin supports three checkpointing modes: naïve synchronous checkpointing, an asynchronous variation of Cao et al.'s Zig-Zag algorithm [10], and an asynchronous snapshot mode that is supported only when the storage layer supports full multiversioning.

The first mode uses the redundancy inherent in an actively replicated system in order to create a system checkpoint. The system can periodically freeze an entire replica and produces a full-versioned snapshot of the system. Since this only happens at one snapshot at a time, the period during which the replica is unavailable is not seen by the client.

One problem with this approach is that the replica taking the checkpoint may fall significantly behind other replicas, which can be problematic if it is called into action due to a hardware failure in another replica. In addition, it may take the replica significant time for it to catch back up to other replicas, especially in a heavily loaded system.

Calvin's second checkpointing mode is closely based on Cao et al.'s Zig-Zag algorithm [10]. Zig-Zag stores two copies of each record in given datastore, $AS[K]_0$ and $AS[K]_1$, plus two additional bits per record, $MR[K]$ and $MW[K]$ (where K is the key of the record). $MR[K]$ specifies which record version should be used when reading record K from the database, and $MW[K]$ specifies which version to overwrite when updating record K. So new values of record K are always written to $AS[K]_{MW[K]}$, and $MR[K]$ is set equal to $MW[K]$ each time K is updated.

Each checkpoint period in Zig-Zag begins with setting $MW[K]$ equal to $\neg MR[K]$ for all keys K in the database during a physical point of consistency in which the database is entirely quiesced. Thus $AS[K]_{MW[K]}$ always stores the latest version of the record,

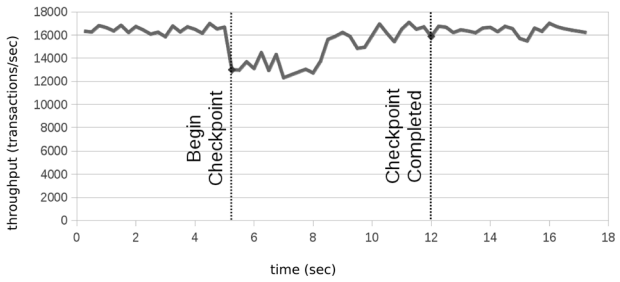

Figure 3: Throughput over time during a typical checkpointing period using Calvin's modified Zig-Zag scheme.

and $AS[K]_{\neg MW[K]}$ always stores the last value written prior to the beginning of the most recent the checkpoint period. An asynchronous checkpointing thread can therefore go through every key K, logging $AS[K]_{\neg MW[K]}$ to disk without having to worry about the record being clobbered.

Taking advantage of Calvin's global serial order, we implemented a variant of Zig-Zag that does *not* require quiescing the database to create a physical point of consistency. Instead, Calvin captures a snapshot with respect to a *virtual* point of consistency, which is simply a pre-specified point in the global serial order. When a virtual point of consistency approaches, Calvin's storage layer begins keeping two versions of each record in the storage system—a "before" version, which can only be updated by transactions that precede the virtual point of consistency, and an "after" version, which is written to by transactions that appear after the virtual point of consistency. Once all transactions preceding the virtual point of consistency have completed executing, the "before" versions of each record are effectively immutable, and an asynchronous checkpointing thread can begin checkpointing them to disk. Once the checkpoint is completed, any duplicate versions are garbage-collected: all records that have both a "before" version and an "after" version discard their "before" versions, so that only one record is kept of each version until the next checkpointing period begins.

Whereas Calvin's first checkpointing mode described above involves stopping transaction execution entirely for the duration of the checkpoint, this scheme incurs only moderate overhead while the asynchronous checkpointing thread is active. Figure 3 shows Calvin's maximum throughput over time during a typical checkpoint capture period. This measurement was taken on a single-machine Calvin deployment running our microbenchmark under low contention (see section 6 for more on our experimental setup).

Although there is some reduction in total throughput due to (a) the CPU cost of acquiring the checkpoint and (b) a small amount of latch contention when accessing records, writing stable values to storage asynchronously does not increase lock contention or transaction latency.

Calvin is also able to take advantage of storage engines that explicitly track all recent versions of each record in addition to the current version. Multiversion storage engines allow read-only queries to be executed without acquiring any locks, reducing overall contention and total concurrency-control overhead at the cost of increased memory usage. When running in this mode, Calvin's checkpointing scheme takes the form of an ordinary "SELECT *" query over all records, where the query's result is logged to a file on disk rather than returned to a client.

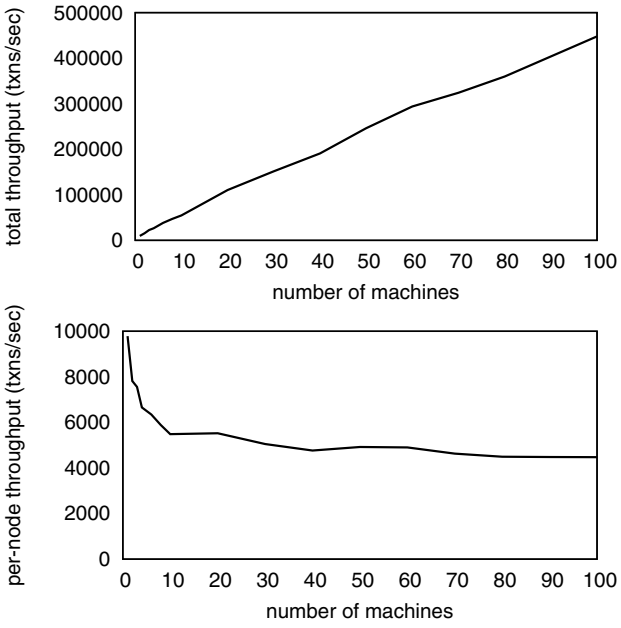

Figure 4: Total and per-node TPC-C (100% New Order) throughput, varying deployment size.

6. PERFORMANCE AND SCALABILITY

To investigate Calvin's performance and scalability characteristics under a variety of conditions, we ran a number of experiments using two benchmarks: the TPC-C benchmark and a Microbenchmark we created in order to have more control over how benchmark parameters are varied. Except where otherwise noted, all experiments were run on Amazon EC2 using High-CPU/Extra-Large instances, which promise 7GB of memory and 20 EC2 Compute Units—8 virtual cores with 2.5 EC2 Compute Units each[4].

6.1 TPC-C benchmark

The TPC-C benchmark consists of several classes of transactions, but the bulk of the workload—including almost all distributed transactions that require high isolation—is made up by the New Order transaction, which simulates a customer placing an order on an eCommerce application. Since the focus of our experiments are on distributed transactions, we limited our TPC-C implementation to only New Order transactions. We would expect, however, to achieve similar performance and scalability results if we were to run the complete TPC-C benchmark.

Figure 4 shows total and per-machine throughput (TPC-C New Order transactions executed per second) as a function of the number of Calvin nodes, each of which stores a database partition containing 10 TPC-C warehouses. To fully investigate Calvin's handling of distributed transactions, multi-warehouse New Order transactions (about 10% of total New Order transactions) always access a second warehouse that is *not* on the same machine as the first.

Because each partition contains 10 warehouses and New Order updates one of 10 "districts" for some warehouse, at most 100 New Order transactions can be executing concurrently at any machine (since there are no more than 100 unique districts per partition, and each New Order transaction requires an exclusive lock on a

[4]Each EC2 Compute Unit provides the roughly the CPU capacity of a 1.0 to 1.2 GHz 2007 Opteron or 2007 Xeon processor.

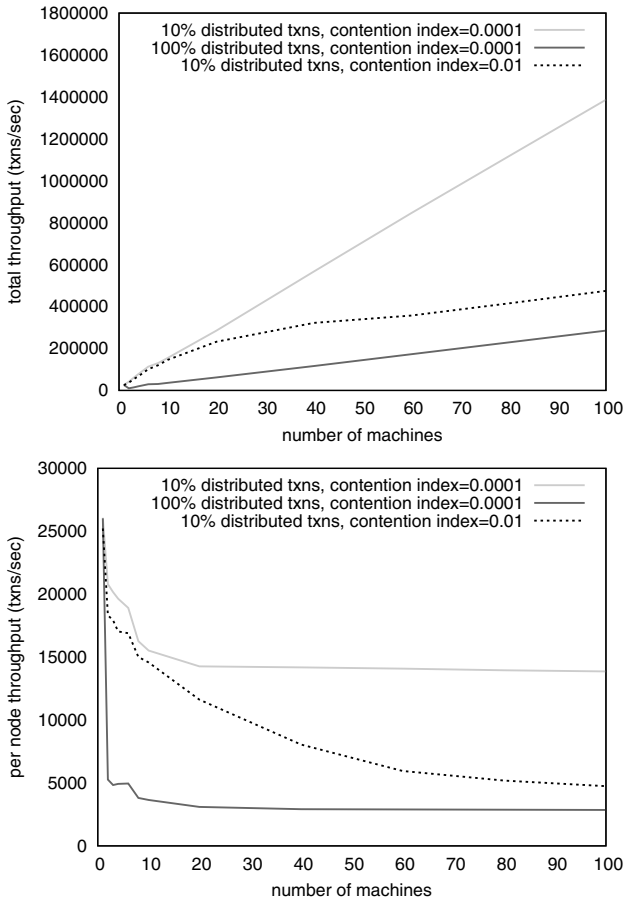

Figure 5: Total and per-node microbenchmark throughput, varying deployment size.

district). Therefore, it is critical that the time that locks are held is minimized, since the throughput of the system is limited by how fast these 100 concurrent transactions complete (and release locks) so that new transactions can grab exclusive locks on the districts and get started.

If Calvin were to hold locks during an agreement protocol such as two-phase commit for distributed New Order transactions, throughput would be severely limited (a detailed comparison to a traditional system implementing two-phase commit is given in section 6.3). Without the agreement protocol, Calvin is able to achieve around 5000 transactions per second per node in clusters larger than 10 nodes, and scales linearly. (The reason why Calvin achieves more transactions per second per node on smaller clusters is discussed in the next section.) Our Calvin implementation is therefore able to achieve nearly half a million TPC-C transactions per second on a 100 node cluster. It is notable that the present TPC-C world record holder (Oracle) runs 504,161 New Order transactions per second, despite running on much higher end hardware than the machines we used for our experiments [4].

6.2 Microbenchmark experiments

To more precisely examine the costs incurred when combining distributed transactions and high contention, we implemented a Microbenchmark that shares some characteristics with TPC-C's New Order transaction, while reducing overall overhead and allowing finer adjustments to the workload. Each transaction in the benchmark reads 10 records, performs a constraint check on the result, and updates a counter at each record if and only if the constraint check passed. Of the 10 records accessed by the microbenchmark transaction, one is chosen from a small set of "hot" records[5], and the rest are chosen from a very much larger set of records—except when a microbenchmark transaction spans two machines, in which case it accesses one "hot" record on *each* machine participating in the transaction. By varying the number of "hot" records, we can finely tune contention. In the subsequent discussion, we use the term *contention index* to refer to the fraction of the total "hot" records that are updated when a transaction executes at a particular machine. A contention index of 0.001 therefore means that each transaction chooses one out of one thousand "hot" records to update at each participating machine (i.e. at most 1000 transactions could ever be executing concurrently), while a contention index of 1 would mean that every transaction touches *all* "hot" records (i.e. transactions must be executed completely serially).

Figure 5 shows experiments in which we scaled the Microbenchmark to 100 Calvin nodes under different contention settings and with varying numbers of distributed transactions. When adding machines under very low contention (contention index $= 0.0001$), throughput per node drops to a stable amount by around 10 machines and then stays constant, scaling linearly to many nodes. Under higher contention (contention index $= 0.01$, which is similar to TPC-C's contention level), we see a longer, more gradual per-node throughput degradation as machines are added, more slowly approaching a stable amount.

Multiple factors contribute to the shape of this scalability curve in Calvin. In all cases, the sharp drop-off between one machine and two machines is a result of the CPU cost of additional work that must be performed for every multipartition transaction:

- Serializing and deserializing remote read results.
- Additional context switching between transactions waiting to receive remote read results.
- Setting up, executing, and cleaning up after the transaction at *all* participating machines, even though it is counted *only once* in total throughput.

After this initial drop-off, the reason for further decline as more nodes are added—even when both the contention and the number of machines participating in any distributed transaction are held constant—is quite subtle. Suppose, under a high contention workload, that machine A starts executing a distributed transaction that requires a remote read from machine B, but B hasn't gotten to that transaction yet (B may still be working on earlier transactions in the sequence, and it can not start working on the transaction until locks have been acquired for all previous transactions in the sequence). Machine A may be able to begin executing some other non-conflicting transactions, but soon it will simply have to wait for B to catch up before it can commit the pending distributed transaction and execute subsequent conflicting transactions. By this mechanism, there is a limit to how far ahead of or behind the pack any particular machine can get. The higher the contention, the tighter this limit. As machines are added, two things happen:

- **Slow machines.** Not all EC2 instances yield equivalent performance, and sometimes an EC2 user gets stuck with a slow

[5]Note that this is a different use of the term "hot" than that used in the discussion of caching in our earlier discussion of memory- vs. disk-based storage engines.

instance. Since the experimental results shown in Figure 5 were obtained using the same EC2 instances for all three lines and all three lines show a sudden drop between 6 and 8 machines, it is clear that a slightly slow machine was added when we went from 6 nodes to 8 nodes.

- **Execution progress skew.** Every machine occasionally gets slightly ahead of or behind others due to many factors, such as OS thread scheduling, variable network latencies, and random variations in contention between sequences of transactions. The more machines there are, the more likely at any given time there will be at least one that is slightly behind for some reason.

The sensitivity of overall system throughput to execution progress skew is strongly dependent on two factors:

- **Number of machines.** The fewer machines there are in the cluster, the more each additional machine will increase skew. For example, suppose each of n machines spends some fraction k of the time contributing to execution progress skew (i.e. falling behind the pack). Then at each instant there would be a $1-(1-k)^n$ chance that at least one machine is slowing the system down. As n grows, this probability approaches 1, and each additional machine has less and less of a skewing effect.

- **Level of contention.** The higher the contention rate, the more likely each machine's random slowdowns will be to cause *other* machines to have to slow their execution as well. Under low contention (contention index $= 0.0001$), we see per-node throughput decline sharply only when adding the first few machines, then flatten out at around 10 nodes, since the diminishing increases in execution progress skew have relatively little effect on total throughput. Under higher contention (contention index $= 0.01$), we see an even sharper initial drop, and then it takes many more machines being added before the curve begins to flatten, since even small incremental increases in the level of execution progress skew can have a significant effect on throughput.

6.3 Handling high contention

Most real-world workloads have low contention most of the time, but the appearance of small numbers of *extremely* hot data items is not infrequent. We therefore experimented with Calvin under the kind of workload that we believe is the primary reason that so few practical systems attempt to support distributed transactions: combining many multipartition transactions with very high contention. In this experiment we therefore do not focus on the entirety of a realistic workload, but instead we consider only the subset of a workload consisting of high-contention multipartition transactions. Other transactions can still conflict with these high-conflict transactions (on records besides those that are very hot), so the throughput of this subset of an (otherwise easily scalable) workload may be tightly coupled to overall system throughput.

Figure 6 shows the factor by which 4-node and 8-node Calvin systems are slowed down (compared to running a perfectly partitionable, low-contention version of the same workload) while running 100% multipartition transactions, depending on contention index. Recall that contention index is the fraction of the total set of hot records locked by each transaction, so a contention index of 0.01 means that up to 100 transactions can execute concurrently, while a contention index of 1 forces transactions to run completely serially.

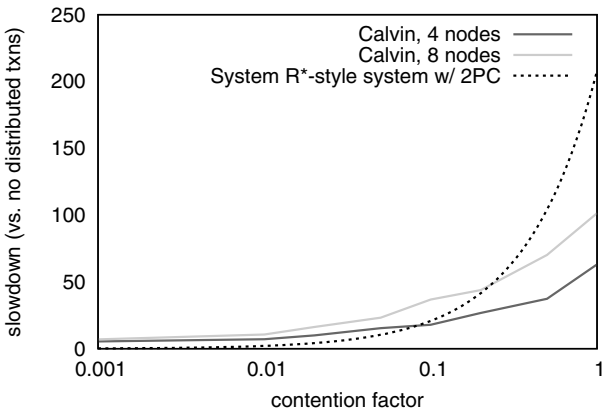

Figure 6: Slowdown for 100% multipartition workloads, varying contention index.

Because modern implementations of distributed systems do not implement System R*-style distributed transactions with two-phase commit, and comparisons with any earlier-generation systems would not be an apples-to-apples comparison, we include for comparison a simple model of the contention-based slowdown that would be incurred by this type of system. We assume that in the non-multipartition, low-contention case this system would get similar throughput to Calvin (about 27000 microbenchmark transactions per second per machine). To compute the slowdown caused by multipartition transactions, we consider the extended contention footprint caused by two-phase commit. Since given a contention index C at most $1/C$ transactions can execute concurrently, a system running 2PC at commit time can never execute more than

$$\frac{1}{C * D_{2PC}}$$

total transactions per second where where D_{2PC} is the duration of the two-phase commit protocol.

Typical round-trip ping latency between nodes in the same EC2 data center is around 1 ms, but including delays of message multiplexing, serialization/deserialization, and thread scheduling, one-way latencies in our system between transaction execution threads are almost never less than 2 ms, and usually longer. In our model of a system similar in overhead to Calvin, we therefore expect to locks to be held for approximately 8ms on each distributed transaction. Note that this model is somewhat naïve since the contention footprint of a transaction is assumed to include nothing but the latency of two-phase commit. Other factors that contribute to Calvin's actual slowdown are completely ignored in this model, including:

- CPU costs of multipartition transactions

- Latency of reaching a local commit/abort decision before starting 2PC (which may require additional remote reads in a real system)

- Execution progress skew (all nodes are assumed to begin execution of each transaction and the ensuing 2PC in perfect lockstep)

Therefore, the model does not establish a specific comparison point for our system, but a strong lower bound on the slowdown for such a system. In an actual System R*-style system, one might expect

to see considerably more slowdown than predicted by this model in the context of high-contention distributed transactions.

There are two very notable features in the results shown in Figure 6. First, under low contention, Calvin gets the same approximately 5x to 7x slowdown—from 27000 to about 5000 (4 nodes) or 4000 (8 nodes) transactions per second—as seen in the previous experiment going from 1 machine to 4 or 8. For all contention levels examined in this experiment, the difference in throughput between the 4-node and 8-node cases is a result of increased skew in workload execution progress between the different nodes; as one would predict, the detrimental effect of this skew to throughput is significantly worse at higher contention levels.

Second, as expected, at very high contentions, even though we ignore a number of the expected costs, the model of the system running two-phase commit incurs significantly more slowdown than Calvin. This is evidence that (a) the distributed commit protocol is a major factor behind the decision for most modern distributed system not to support ACID transactions and (b) Calvin alleviates this issue.

7. RELATED WORK

One key contribution of the Calvin architecture is that it features active replication, where the same transactional input is sent to multiple replicas, each of which processes transactional input in a deterministic manner so as to avoid diverging. There have been several related attempts to actively replicate database systems in this way. Pacitti et al. [23], Whitney et al. [29], Stonebraker et al.[27], and Jones et al. [16] all propose performing transactional processing in a distributed database without concurrency control by executing transactions serially—and therefore equivalently to a *known* serial order—in a single thread on each node (where a node in some cases can be a single CPU core in a multi-core server [27]). By executing transactions serially, nondeterminism due to thread scheduling of concurrent transactions is eliminated, and active replication is easier to achieve. However, serializing transactions can limit transactional throughput, since if a transaction stalls (e.g. for a network read), other transactions are unable to take over. Calvin enables concurrent transactions while still ensuring logical equivalence to a given serial order. Furthermore, although these systems choose a serial order in advance of execution, adherence to that order is not as strictly enforced as in Calvin (e.g. transactions can be aborted due to hardware failures), so two-phase commit is still required for distributed transactions.

Each of the above works implements a system component analogous to Calvin's sequencing layer that chooses the serial order. Calvin's sequencer design most closely resembles the H-Store design [27], in which clients can submit transactions to any node in the cluster. Synchronization of inputs between replicas differs, however, in that Calvin can use either asynchronous (log-shipping) replication or Paxos-based, strongly consistent synchronous replication, while H-Store replicates inputs by stalling transactions by the expected network latency of sending a transaction to a replica, and then using a deterministic scheme for transaction ordering assuming all transactions arrive from all replicas within this time window.

Bernstein et al.'s Hyder [8] bears conceptual similarities to Calvin despite extremely different architectural designs and approaches to achieving high scalability. In Hyder, transactions submit their "intentions"—buffered writes—after executing based on a view of the database obtained from a recent snapshot. The intentions are composed into a global order and processed by a deterministic "meld" function, which determines what transactions to commit and what transactions must be aborted (for example due to a data update that invalidated the transaction's view of the database *after* the transaction executed, but *before* the meld function validated the transaction). Hyder's globally-ordered log of things-to-attempt-deterministically is comprised of the after-effects of transactions, whereas the analogous log in Calvin contains unexecuted transaction requests. However, Hyder's optimistic scheme is conceptually very similar to the Optimistic Lock Location Prediction scheme (OLLP) discussed in section 3.2.1. OLLP's "reconnaissance" queries determine the transactional inputs, which are deterministically validated at "actual" transaction execution time in the same optimistic manner that Hyder's meld function deterministically validates transactional results.

Lomet et al. propose "unbundling" transaction processing system components in a cloud setting in a manner similar to Calvin's separation of different stages of the pipeline into different subsystems [21]. Although Lomet et al.'s concurrency control and replication mechanisms do not resemble Calvin's, both systems separate the "Transactional Component" (scheduling layer) from the "Data Component" (storage layer) to allow arbitrary storage backends to serve the transaction processing system depending on the needs of the application. Calvin also takes the unbundling one step further, separating out the sequencing layer, which handles data replication.

Google's Megastore [7] and IBM's Spinnaker [25] recently pioneered the use of the Paxos algorithm [18, 19] for strongly consistent data replication in modern, high-volume transactional databases (although Paxos and its variants are widely used to reach synchronous agreement in countless other applications). Like Calvin, Spinnaker uses ZooKeeper [15] for its Paxos implementation. Since they are not deterministic systems, both Megastore and Spinnaker must use Paxos to replicate transactional effects, whereas Calvin only has to use Paxos to replicate transactional inputs.

8. FUTURE WORK

In its current implementation, Calvin handles hardware failures by recovering the crashed machine from its most recent complete snapshot and then replaying all more recent transactions. Since other nodes within the same replica may depend on remote reads from the afflicted machine, however, throughput in the rest of the replica is apt to slow or halt until recovery is complete.

In the future we intend to develop a more seamless failover system. For example, failures could be made completely invisible with the following simple technique. The set of all replicas can be divided into replication subgroups—pairs or trios of replicas located near one another, generally on the same local area network. Outgoing messages related to multipartition transaction execution at a database node A in one replica are sent not only to the intended node B within the same replica, but also to every replica of node B within the replication subgroup—just in case one of the subgroup's node A replicas has failed. This redundancy technique comes with various tradeoffs and would not be implemented if inter-partition network communication threatened to be a bottleneck (especially since active replication in deterministic systems already provides high availability), but it illustrates a way of achieving a highly "hiccup-free" system in the face of failures.

A good compromise between these two approaches might be to integrate a component that monitor each node's status, could detect failures and carefully orchestrate quicker failover for a replica with a failed node by directing other replicas of the afflicted machine to forward their remote read messages appropriately. Such a component would also be well-situated to oversee load-balancing of read-only queries, dynamic data migration and repartitioning, and load monitoring.

9. CONCLUSIONS

This paper presents Calvin, a transaction processing and replication layer designed to transform a generic, non-transactional, unreplicated data store into a fully ACID, consistently replicated distributed database system. Calvin supports horizontal scalability of the database and unconstrained ACID-compliant distributed transactions while supporting both asynchronous and Paxos-based synchronous replication, both within a single data center and across geographically separated data centers. By using a deterministic framework, Calvin is able to eliminate distributed commit protocols, the largest scalability impediment of modern distributed systems. Consequently, Calvin scales near-linearly and has achieved near-world record transactional throughput on a simplified TPC-C benchmark.

10. ACKNOWLEDGMENTS

This work was sponsored by the NSF under grants IIS-0845643 and IIS-0844480. Kun Ren is supported by National Natural Science Foundation of China under Grant 61033007 and National 973 project under Grant 2012CB316203. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation (NSF) or the National Natural Science foundation of China.

11. REFERENCES

[1] Amazon simpledb. http://aws.amazon.com/simpledb/.

[2] Project voldemort. http://project-voldemort.com/.

[3] Riak. http://wiki.basho.com/riak.html.

[4] Transaction processing performance council. http://www.tpc.org/tpcc/.

[5] D. Abadi. Replication and the latency-consistency tradeoff. http://dbmsmusings.blogspot.com/2011/12/replication-and-latency-consistency.html.

[6] J. C. Anderson, J. Lehnardt, and N. Slater. *CouchDB: The Definitive Guide*. 2010.

[7] J. Baker, C. Bond, J. Corbett, J. J. Furman, A. Khorlin, J. Larson, J.-M. Leon, Y. Li, A. Lloyd, and V. Yushprakh. Megastore: Providing scalable, highly available storage for interactive services. In *CIDR*, 2011.

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

[9] D. Campbell, G. Kakivaya, and N. Ellis. Extreme scale with full sql language support in microsoft sql azure. In *SIGMOD*, 2010.

[10] T. Cao, M. Vaz Salles, B. Sowell, Y. Yue, A. Demers, J. Gehrke, and W. White. Fast checkpoint recovery algorithms for frequently consistent applications. In *SIGMOD*, 2011.

[11] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: a distributed storage system for structured data. In *OSDI*, 2006.

[12] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. Pnuts: Yahoo!'s hosted data serving platform. *VLDB*, 2008.

[13] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. *SIGOPS*, 2007.

[14] S. Gilbert and N. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 2002.

[15] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *In USENIX Annual Technical Conference*.

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

[17] A. Lakshman and P. Malik. Cassandra: structured storage system on a p2p network. In *PODC*, 2009.

[18] L. Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 1998.

[19] L. Lamport. Paxos made simple. *ACM SIGACT News*, 2001.

[20] D. Lomet and M. F. Mokbel. Locking key ranges with unbundled transaction services. *VLDB*, 2009.

[21] D. B. Lomet, A. Fekete, G. Weikum, and M. J. Zwilling. Unbundling transaction services in the cloud. In *CIDR*, 2009.

[22] C. Mohan, B. G. Lindsay, and R. Obermarck. Transaction management in the r* distributed database management system. *ACM Trans. Database Syst.*, 1986.

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

[24] E. Plugge, T. Hawkins, and P. Membrey. *The Definitive Guide to MongoDB: The NoSQL Database for Cloud and Desktop Computing*. 2010.

[25] J. Rao, E. J. Shekita, and S. Tata. Using paxos to build a scalable, consistent, and highly available datastore. *VLDB*, 2011.

[26] M. Seltzer. Oracle nosql database. In *Oracle White Paper*, 2011.

[27] 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*, 2007.

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

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